Computers & Security 87 (2019) 101597

Contents lists available at ScienceDirect

Computers & Security

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

Towards a reliable firewall for software-defined networks

Hongxin Hu[a,*], Wonkyu Han[b], Sukwha Kyung[b], Juan Wang[d], Gail-Joon Ahn[b], Ziming Zhao[c], Hongda Li[a]

[a] School of Computing, Clemson University, Clemson, SC 29634, USA
[b] Center for Cybersecurity and Digital Forensics (CDF), Arizona State University, Tempe, AZ 85287, USA
[c] Golisano College of Computing and Information Sciences, Rochester Institute of Technology, Rochester, NY 14623, USA
[d] Computer School, Wuhan University, Wuhan 430072, China

ARTICLE INFO

Article history:
Received 13 December 2018
Accepted 2 April 2019
Available online 31 August 2019

Keywords:
Firewalls
Policy violation
Software-Defined networking
Openflow
Network security

ABSTRACT

Software-Defined Networking (SDN) is an emerging paradigm in networking where network control plane is decoupled from forwarding plane through programmable control. OpenFlow – the most popular SDN platform – introduces significant granularity, visibility and flexibility to networking, but at the same time brings forth new security challenges. One of the fundamental challenges is to build a reliable firewall for protecting OpenFlow networks where network states and traffic are frequently changed. To address this challenge, we introduce FlowMon, an OpenFlow-based firewall, to support network-wide access control by facilitating not only accurate violation detection but also effective violation resolution in dynamic OpenFlow networks. FlowMon detects firewall policy violations by checking flow path space against firewall authorization space when a flow entry or firewall rule is inserted, modified, or deleted. In particular, FlowMon conducts automatic and real-time violation resolutions with the help of several innovative resolution strategies applied to diverse network update situations. We also implement a prototype of FlowMon in Floodlight. Our experimental results demonstrate FlowMon effectively addresses violations in a real-world network topology, and produces manageable performance overhead with effective violation detection and resolution.

Published by Elsevier Ltd.

1. Introduction

Over the past few years, Software-Defined Networking (SDN) has evolved from purely an idea (Casado et al., 2007; 2006; Greenberg et al., 2005) to a new paradigm that various networking vendors are not only embracing, but also pursuing as their model for future enterprise network management. OpenFlow (McKeown et al., 2008), the de-facto standard protocol for SDN, essentially separates the control plane and the data plane of a network device, and enables the network control to become directly programmable as well as the underlying infrastructure to be abstracted for network applications. With OpenFlow, only the data plane exists in the network device, and all control decisions are made on the logically-centralized controller.

One primary goal of SDN is to enable various network applications, which are also called network services or functions, to run on the controller to manage the network directly by configuring packet-handling mechanisms in underlying devices. Consequently, when enterprises adopt OpenFlow for their networks, it is virtually inevitable that legacy security applications such as firewalls and intrusion detection and prevention systems (IDS/IPS) have to be migrated to OpenFlow-based networks by re-designing and implementing them as compatible security applications. In this paper, we focus on addressing the challenges of designing and implementing a *reliable* firewall application for OpenFlow-based networks.

Firewalls are the most widely deployed security mechanism in many businesses and institutions. A conventional firewall sits on the border between a private network and the Internet and examines all incoming and outgoing packets to defend against attacks and unauthorized access. However, one key assumption under this traditional model is that all insiders of the protected network are trusted, since internal traffic is not monitored and cannot be filtered by the firewall (Ioannidis et al., 2000). The assumption has been invalid for a long time, because malicious insiders can easily launch attacks on others in the network by circumventing security mechanisms (Schultz, 2002). With OpenFlow, such a problem could be alleviated, since OpenFlow offers finer level of control granular-

* Corresponding author.
E-mail addresses: hongxih@clemson.edu (H. Hu), whan7@asu.edu (W. Han), skyung1@asu.edu (S. Kyung), jwang@whu.edu.cn (J. Wang), gahn@asu.edu (G.-J. Ahn), zxzics@rit.edu (Z. Zhao), hongdal@clemson.edu (H. Li).

https://doi.org/10.1016/j.cose.2019.101597
0167-4048/Published by Elsevier Ltd.

ity so that enforcement can be placed at any entry points of traffic flows in a network.

Unfortunately, OpenFlow also brings great challenges for designing firewall applications in emerging SDNs. First, OpenFlow allows various `Set-Field` actions, which can rewrite the values of header fields in packets. Such a feature can, in fact, significantly increase the usefulness of an OpenFlow implementation. For example, a load balancer application may need to dynamically change flow paths and destinations. However, *adversaries* could also take advantage of this feature to strategically modify flow rules (i.e., entries on the flow tables) that would circumvent network security mechanisms (for example, firewalls).[1] Second, in an OpenFlow network, network states are dynamically updated and configurations are frequently changed. Thus, simply checking *flow packet violation* by monitoring packet-in behaviors is not effective, because *flow policy violation*, which is a violation induced by proactive installation or modification of flow rules, network states or configurations, should also be detected and resolved in real time as well. Last but not least, when a security violation is detected, a firewall cannot simply reject the new flow rule(s) or remove the already-existing flow rule(s) that causes the violation. In OpenFlow, multiple traffic flows may match the same rule. Also, OpenFlow allows using wildcard rules to define a flow. Because of these characteristics of OpenFlow, if only *partial* packets violate the firewall policy, eliminating the matching flow rule may drop legal traffic which in turn could encumber the availability and utility of network services.

An exemplar firewall application based on OpenFlow is introduced in Floodlight (Floodlight), a popular open-source SDN controller, which enforces security rules against traffic flows by monitoring all packet-in behaviors in the network. Nevertheless, this preliminary implementation only inspects a traffic flow at its *ingress* switch and lacks a capability to actively monitor packet modifications. In other words, once a flow passes the ingress switch, dynamic modifications of the flow cannot be further inspected by the firewall. Also, it can only examine violations when a *new* flow comes in the network, but cannot check any other network updates.

In this paper, we propose a new firewall application, FlowMon, which is designed to facilitate not only accurate detection but also effective resolution of firewall policy violations, and support network-wide access control in dynamic OpenFlow networks. FlowMon detects violations by examining *flow path space* against *firewall authorization space*. The violation detection approach in FlowMon is capable of tracking flow paths in the entire network and checking rule dependencies in both flow table (Kazemian et al., 2013) and firewall policies (Yuan et al., 2006). Besides, FlowMon can detect violations dynamically when network states or configurations are changed. In addition, we introduce a flexible violation resolution framework in FlowMon to enable an automatic and real-time violation resolution, which have not been addressed by existing approaches for SDNs (For example, Kazemian et al., 2013; 2012; Khurshid et al., 2013; Mai et al., 2011). More specifically, we introduce four different resolution strategies, namely *dependency breaking, update rejecting, flow removing*, and *packet blocking*. We demonstrate that these resolution methods can resolve the detected violations, despite of diverse update situations in both flow entries and firewall rules. In order to ensure real-time response in FlowMon, we also address several optimization considerations in the FlowMon design.

The major contributions of this paper are summarized as follows:

- We present security challenges and design requirements in building a firewall application for OpenFlow networks with respect to both *packet modifications* and *rule dependencies* in flow tables and firewall policies.
- We propose a systematic solution for designing an OpenFlow-based firewall application that enables network-wide access control in dynamic OpenFlow networks. Our design addresses challenges created by the *inter-reaction* of flow path and firewall authorization space. Our design facilitates not only accurate detection but also automatic and real-time resolution of firewall policy violations in OpenFlow networks.
- We provide a prototype implementation of FlowMon in an open SDN controller. We evaluate FlowMon using a real-world network topology and emulated OpenFlow network. Our experimental results show that FlowMon has negligible performance overhead to enable real-time violation detection and resolution.

This paper is organized as follows. We first overview related work in Section 2. Section 3 explains the security challenges and design requirements in constructing an OpenFlow-based firewall application. Section 4 presents the design of FlowMon in detail. We address the implementation and evaluation of FlowMon in Section 5. Section 6 describes several important issues related to the design of firewall for OpenFlow-based networks. Finally, we conclude this paper in Section 7.

2. Related work

Several recent efforts are devoted to address various security challenges in SDN, such as topology poisoning prevention (Hong et al., 2015; Khan et al., 2017; Sasan and Salehi, 2017), DDoS attack detection (Alshamrani et al., 2017; Jantila and Chaipah, 2016; Mousavi and St-Hilaire, 2015), vulnerability assessment (Benton et al., 2013; Kreutz et al., 2013; Lee et al., 2017), and saturation attack mitigation (Shin et al., 2013b; Yoon et al., 2017), in SDNs. Differentiating from those work, our work focuses on exploring how to build reliable firewalls for SDNs.

Floodlight contains a firewall application (Floodlight) where each packet-in behavior triggered by the first packet of a traffic flow is matched against the set of existing firewall rules that allow or deny a flow at its ingress switch. However, such a primitive implementation of OpenFlow-based firewall application suffers from a couple of limitations as mentioned briefly in Section 1. Pyretic (Monsanto et al., 2013) is introduced as a higher-level language in the Frenetic Project (Frenetic) that allows SDN programmers to write modular network applications, including firewall application. Pyretic's sequential composition operators could potentially resolve *direct* policy conflicts by compiling conflicting policies into a prioritized rule set. However, Pyretic cannot discover and resolve *indirect* security violations caused by dynamic packet modifications without a flow tracking mechanism (Fayazbakhsh et al., 2013). FortNOX (Porras et al., 2012; 2015) is proposed as a software extension aiming to provide security constraint enforcement for OpenFlow controllers, being able to identify *indirect* security violations. However, we cannot directly adopt the approach introduced in FortNOX to design our firewall application for several reasons. On one hand, the rule conflict analysis algorithm provided by FortNOX records rule relations in alias sets, which are unable to accurately track all flows. In particular, the conflict detection algorithm in FortNOX only conducts *pairwise* conflict analysis between new flow rule(s) and each single security constraint without considering *rule dependencies* within flow tables (Kazemian et al., 2013; Khurshid et al., 2013) and among security constraints (represented as a firewall policy in our approach) (Hari et al., 2000; Yuan et al., 2006). On the other hand, when FortNOX detects a security violation caused by new rule(s) installed by a non-security application, it simply rejects the rule(s) without offering a fine-grained violation resolution.

[1] We further articulate such scenarios in Section 3.2.

Our prior work introduces a conceptual framework, FlowGuard, for building SDN firewalls (Hu et al., 2014b). However, several practical questions still remain: (1) major challenges caused by the feature of packet modification in OpenFlow-based networks; (2) efficient mechanisms to enable *real-time* violation detection and resolution in SDN firewall applications; (3) practical benefits of violation detection and resolution including performance and scalability concerns. In contrast, FlowMon provides a comprehensive solution to address bypass threads when designing SDN firewalls for dynamic OpenFlow-based networks.

A few verification tools (Al-Shaer and Al-Haj, 2010; Kazemian et al., 2013; 2012; Khurshid et al., 2013; Mai et al., 2011) for checking network invariants and policy correctness in OpenFlow networks have been proposed. Anteater (Mai et al., 2011) detects violations of network invariants using a SAT solver through transferring the data-plane information to boolean expressions and converting network invariants into instances of SAT problem. FlowChecker (Al-Shaer and Al-Haj, 2010) translates network policies into boolean expressions and uses Binary Decision Diagram (BDD) to model the network state for checking network invariants. However, both Anteater and FlowChecker are static in nature and could not scale well to dynamic changes in the network. VeriFlow (Khurshid et al., 2013) and NetPlumber (Kazemian et al., 2013) are capable of checking the compliance of network updates with specified invariants in real time. VeriFlow uses graph search techniques to verify network-wide invariants and deals with dynamic changes. NetPlumber utilizes Header Space Analysis (HSA) (Kazemian et al., 2012) in an incremental manner to ensure real-time response for checking network policies through building a dependency graph. Even though these tools can be potentially used to detect firewall policy violations, they are only able to simply raise alarms to indicate possible violations to users, but cannot provide an automatic and real-time violation resolution. Also, they ignore rule dependencies within security constraints, such as firewall policies, for compliance checking.

Policy verification tools discussed above are able to check network reachability and potentially utilized for tracking flow paths in OpenFlow networks. However, Anteater (Mai et al., 2011) and FlowChecker (Al-Shaer and Al-Haj, 2010) are indeed offline systems and cannot be applied for real-time flow tracking. VeriFlow (Khurshid et al., 2013) can perform reachability checking in real time, but it does not support dynamic packet modifications. Another option for flow tracking would be FlowTags (Fayazbakhsh et al., 2013), which can additionally deal with dynamic transformations in the presence of legacy middleboxes (for example, proxies). However, FlowTags needs to alter existing OpenFlow architecture. In this work, we leverage the mechanism introduced in NetPlumber (Kazemian et al., 2013) to track flow paths for firewall policy violation detection, because NetPlumber provides features that fit for our purposes, such as support for arbitrary header modifications, automatic rule dependency detection, and real-time response.

Numerous firewall algorithms and tools are designed to assist system administrators in managing and analyzing firewall policies (Al-Shaer and Hamed, 2004; Alfaro et al., 2008; Baboescu and Varghese, 2003; Hu et al., 2010; 2012; Yuan et al., 2006). Especially, some works present policy analysis tools with the goal of detecting firewall policy conflicts. Al-Shaer and Hamed (2004) designed a tool called Firewall Policy Advisor to detect pairwise anomalies in firewall rules. Yuan et al. (2006) presented FIREMAN, a toolkit to check for misconfiguration in firewall policies through static analysis. Our previous work (Hu et al., 2010; 2012) introduces FAME, a visualization-based firewall anomaly management environment, for detection and resolution of firewall anomalies. However, existing firewall policy analysis tools only detect policy conflicts *within* a firewall policy and cannot be directly applied to deal with firewall policy violations against flow policies in dynamic OpenFlow networks.

There exist other related works to deal with a set of conflict resolution strategies for access control, including Fundulaki and Marx (2004), Jajodia et al. (1997), and Li et al. (2009). These existing solutions mainly focus on resolving rule conflicts in one type of policy. However, in OpenFlow networks, conflicts between two kinds of policies (i.e., firewall policy and flow policy) must be resolved.

3. Background technologies and challenges

Before introducing the design of FlowMon, we describe the concepts of flow policy and firewall policy in this section. We then review security challenges and design requirements that motivate the features of FlowMon.

3.1. Overview of flow and firewall policies

Flow Policy: In an OpenFlow network, flow rules can be added into flow tables, both *reactively* (generating rules in response to the packets of new flows) and *proactively* (installing rules before packets arrive at the switches) (OpenFlow Switch Specification). In the reactive rule generation, if a switch receives a packet for which no matching rule exists on its flow table, then the switch forwards the packet to the controller for further inspection. The controller determines whether that packet should be allowed and can then install a new *flow policy*, which is a collection of rules installed at switches as flow table entries.[2] The installed flow rules are used for handling future packets of the same type. In the proactive rule installation, the controller or applications are allowed to initiate rules in the network devices before receiving flow packets.

While flow *policy* is a high-level term that defines what should be done with the entire flow, a flow *rule* is a specific entry installed on flow table of a switch. Each flow rule specifies a *pattern* that matches on bits in the packet header, *actions* that are performed on matching packets to describe packet forwarding, packet modification or packet dropping, a *priority* that disambiguates among overlapping patterns, and *timeouts* that allow a switch to delete expired rules.

Firewall Policy: Similar to flow policy, a firewall policy consists of a sequence of rules that define the actions performed on packets that satisfy certain conditions. The rules are specified in the form of ⟨*condition, action*⟩. A *condition* in the rule is composed of a set of fields, which is typically specified in a 5-tuple format that contains *source IP, source port, destination IP, destination port*, and *protocol*, to identify a certain type of packets matched by this rule. The general *action* in a firewall rule is either “allow” or “deny”.

In a firewall policy, multiple rules may overlap, which means one packet may match several rules. Moreover, multiple rules within one policy may conflict, implying that those rules not only overlap each other but also yield different decisions. To resolve policy conflicts, a firewall typically implements a *first-match* resolution mechanism based on the order of rules. In this way, each packet processed by the firewall is mapped to the decision of first rule that the packet matches.

3.2. Security challenges

OpenFlow offers greater flexibility to networking. However, at the same time, its flexibility comes with security challenges. One such challenge is introduced by the feature of *packet modification*

[2] In OpenFlow v1.0, each switch consists of one flow table. However, newer versions of OpenFlow allow every switch contains multiple flow tables.

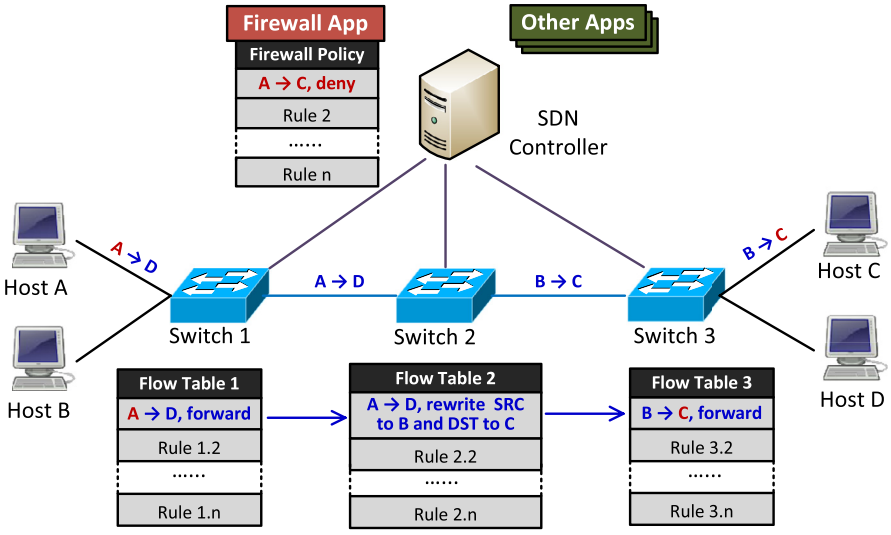


Fig. 1. Firewall is bypassed by a single flow.

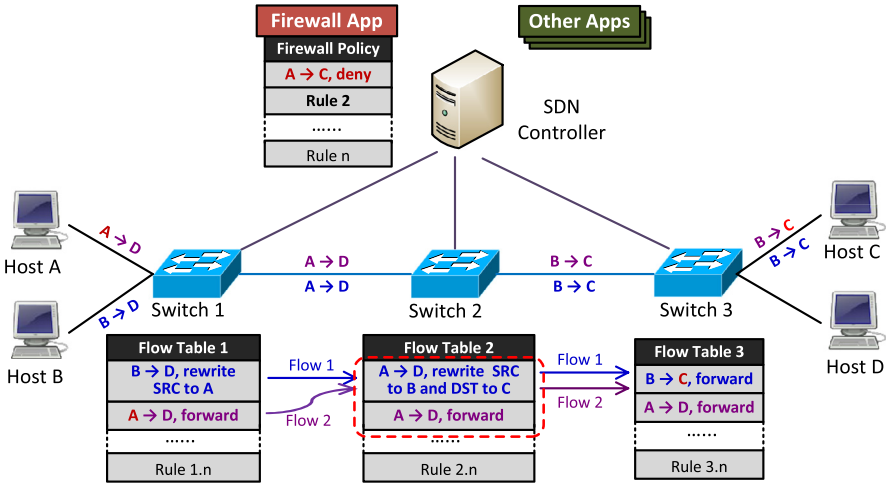


Fig. 2. Firewall is bypassed due to rule dependency.

allowing various `Set-Field` actions that can dynamically change the packet headers. *Adversaries* can leverage this feature to evade network security mechanisms, such as firewalls. Another challenge may arise from *rule dependency* in flow tables and firewall policies. Flow rules may overlap each other in a flow table, indicating intra-table dependency of flow rules (Kazemian et al., 2013). The rules in a firewall policy may overlap as well (Hari et al., 2000; Yuan et al., 2006). These rule dependencies could also be leveraged by *malicious* OpenFlow applications and may cause severe network breaches. We articulate two hypothetical scenarios to elaborate these challenges. To make our discussion concrete, we use an example network shown in Figs. 1 and 2 with three switches, four hosts, and one SDN controller on which a simple firewall application (for example, the Floodlight built-in firewall application) and several other applications are running.

Bypass Scenario 1: The first scenario illustrates firewall rule violation caused by packet modification (see Fig. 1). In this scenario, the firewall application has a rule to deny network packets from *Host A* to *Host C*.[3] Suppose that another application running on the controller establishes a new flow policy, which contains three flow rules installed in flow tables of each switch in the network. The first rule in the flow policy allows to simply forward packets from *Host A* to *Host D*. The second rule installed in *Switch 2* (see Fig. 1) rewrites the source address (SRC) of a packet to *Host B* and the destination address (DST) of the packet to *Host C*. The last rule forwards packets from *Host B* to *Host C*. In this case, if *Host A* sends a packet to *Host D*, the packet will be delivered to *Host C*, which violates the firewall rule. However, if the firewall only inspects the flow at its ingress switch (*Switch 1*) without tracking the entire flow in the network, such a violation cannot be observed by the firewall.

Bypass Scenario 2: The other bypass scenario, caused by rule dependency, is illustrated in Fig. 2. Same as the first scenario, forwarding packets from *Host A* to *Host C* is denied by the firewall policy. Suppose a flow policy for *Flow 1* is installed by an application in the network. The rule installed in *Switch 1* modifies the source address of the matched packets to *Host A*. Since the *original* source of this flow is *Host B* and the *final* destination is *Host C*, the flow policy does not violate the firewall policy. Now, suppose another application installs a new policy, which contains three forwarding rules, for another flow (*Flow 2*) in the network. This policy is allowed by the firewall, since it directly forwards packets from *Host A* to *Host D* and does not violate the firewall policy either. The problem occurs at *Switch 2*, where the flow rule for *Flow 2* is installed with lower priority than the flow rule for *Flow 1*. The two

[3] For brevity, we use the host name to represent the source and destination directly in the example rules.

rules, each of which belonging to different flow policies, overlap each other as they both match packets with *Host A* as the source address and *Host D* as the destination address. Since the priority of flow rule for *Flow 1* is higher than that for *Flow 2*, the header fields of packets belonging to *Flow 2* originally sent from *Host A* to *Host D* are changed and eventually sent to *Host C*. This situation rises a violation of the firewall policy. Thus, even though individual policy defined for different flows (In this case, *Flow 1* and *Flow 2*) does not violate the firewall policy, the *dependency relations* among them may induce violation(s). Moreover, rule dependency could also lead to shadowing over the existing firewall rules (Yuan et al., 2006), where a firewall policy with higher precedence nullifies the other firewall policies, and thus, no violation related to the existing firewall policies can be detected.

The built-in firewall application in Floodlight and Pyretic's composition operators cannot detect and resolve both bypass scenarios discussed above, because they are unable to monitor dynamic packet modifications. FortNOX has a limitation in identifying the violation caused by rule dependency, as the rule conflict analysis algorithm in FortNOX ignores rule dependencies in flow tables and firewall policies.

3.3. Design requirements

Our goal is to design a reliable firewall application that detects and resolves firewall policy violations effectively and efficiently in dynamic OpenFlow networks. To that end, we present a solution that fulfills following design requirements to balance network protection and system performance.

1. *Accuracy.* The firewall application should precisely detect violations caused by traffic modifications, as well as rule dependencies in both flow tables and firewall policies. Also, the identified violations should be effectively resolved with respect to different violation situations, such as *partial* or *entire* violations (See Section 4.1.4).
2. *Flexibility.* The firewall application should have the capability to inspect any network state and configuration updates, which may potentially incur firewall policy violations. In addition, flexible resolution strategies should be provided to address various violations utilizing fine-grained control of SDN.
3. *Efficiency.* It is critical that the firewall application works continuously in a timely fashion, because the state of an OpenFlow-based network generally evolves rapidly. Thus, it naturally requires that the response time of the firewall application should be in real-time to fit in the dynamic OpenFlow-based network. Also, its performance overhead should not affect other network services.

4. FlowMon design

We introduce our design of FlowMon that satisfies the proposed requirements in Section 3.3. We focus on two key functions in FlowMon: violation detection and resolution. At high level, FlowMon detects policy violation by verifying the reachability of nodes in the plumbing graph. Based on the type of detected violation, a resolution method is selected and the respective changes are made in the flow rules, if deemed necessary.

4.1. Violation detection

OpenFlow allows the header fields of flow packets to dynamically change when the packets traverse the network. Thus, to support accurate violation detection, a firewall needs to check violations at the ingress switch of each flow. It should also track the flow path and then clearly identify both the *original* source and *final* destination of each flow in the network. In this section, we first introduce two kinds of flow path classifications and then articulate our violation detection method.

4.1.1. Flow path classification

A flow path is a forwarding path where one or multiple flows can pass through in the network. It consists of a sequence of (switch, rule) pairs and is denoted by:

$$(s_1, r_1) \rightarrow \ldots \rightarrow (s_{n-1}, r_{n-1}) \rightarrow (s_n, r_n).$$

OpenFlow allows modification of packet headers in a flow when it passes through the network. For simplifying the calculations involved, we divide flow paths into two categories: *direct flow path* and *shifted flow path*. In a direct flow path, all rules only perform "forward" action to the matched packets. In a *shifted (or indirect) flow path*, at least one rule enforces `Set-Field` action(s) to change the header fields (for example, IP source and destination) of the matched packets. Fig. 3 shows two examples for these paths. In the direct flow path shown in Fig. 3(a), *Host A* sends packets to *Host B* without any changes.

In the shifted path depicted in Fig. 3(b), the destination of packets sent by *Host A* is changed from *Host B* to *Host C*. In our description of the security threats in Section 3.2, the flow rule violations are mainly caused by *shifted flow path*. Therefore, the accuracy of FlowMon depends on its abilities to track the flows in such flow paths and resolve the detected violations. To track the flows in those paths, we develop a concept of *Shifted Flow Path Graph* (SFPG) by leveraging HSA (see Section 4.1.2), along with a *Flow Tagging* mechanism (see Section 4.2.1). We also evaluate our implementation of FlowMon in Section 5.2 by simulating the violation caused by *shifted flow paths* to demonstrate the correctness of SFPG and accuracy of FlowMon.

4.1.2. Flow path space analysis

Flow Tracking: To support network-wide access control in an OpenFlow network, a firewall needs to figure out both the *original* source address and *final* destination address of each flow in the network through tracking its flow path. Accordingly, we need an effective flow tracking mechanism to identify flow paths. Several existing network invariant verification tools (Kazemian et al., 2013; Khurshid et al., 2013) check network reachability in real time and be potentially used to help find flow paths in OpenFlow networks. We currently leverage NetPlumber (Kazemian et al., 2013) as a basis for building our flow tracking mechanism, since NetPlumber offers several features that can provide flexibility for effective and accurate flow tracking: (1) building on HSA (Kazemian et al., 2012), it uses a geometric model (*Header Space*) of packet processing to provide a uniform and protocol-independent model of the network; (2) it models networking boxes using a switch transfer function, which can transform a received header to a set of packet headers arbitrarily, supporting dynamic packet modifications. NetPlumber describes measures to construct a *plumbing graph*, which represents all next-hop dependencies and intra-table dependencies of flow rules. The plumbing graph captures both the *direct* and *shifted* flow paths in the network automatically. Building on HSA (Kazemian et al., 2012), NetPlumber uses a geometric model (*Header Space*) of packet processing to provide a uniform and protocol-independent model of the network. NetPlumber also models networking boxes using a switch transfer function, which can transform a received header to a set of packet headers arbitrarily, supporting dynamic packets modifications and constructs a plumbing graph, which represents all next-hop dependencies and intra-table dependencies of rules. Through such a plumbing graph, all flow paths including both *direct* and *shifted* flow paths in the network can be automatically captured.

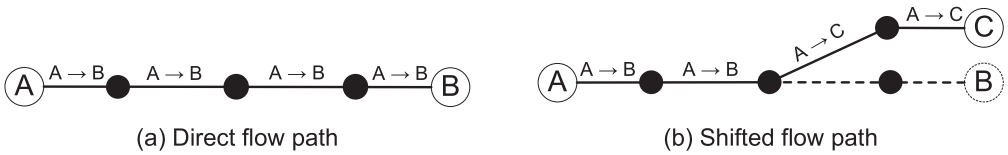


Fig. 3. Examples of direct flow path and shifted flow path.

In a direct flow path, the packet headers remain unchanged when the flow packets pass through it. For checking the firewall policy violations, it is not necessary to track this kind of direct flow path. Thus, violations can be simply identified at the ingress switch. However, for shifted flow paths, the header fields dynamically change while the packets are in route. Violations that can potentially rise in such paths cannot be identified solely at the ingress switch. Therefore, we introduce a concept of *Shifted Flow Path Graph* (SFPG), which is a sub-graph of the plumbing graph and contains all the shifted flow paths. We also include partial direct flow paths that have dependency relations with shifted flow paths as part of the sub-graph, SFPG. Thus, by maintaining and dealing with an SFPG graph when monitoring an OpenFlow network, FlowMon can significantly reduce the overhead involved in the flow tracking process.

Flow Path Space Calculation: The tracked flows represented in *SFPG* graph are used to calculate the header spaces of a flow at different stages while traversing in the network. We only abstract fields required for checking firewall policy violations from the pattern expression of a flow rule to represent a flow path space. Additionally, we reorganize these fields with a (*source address, destination address*) pair, denoted as $[P^s, P^d]$, to specify a *flow path space*. In the context of IP 5-tuple, the source address P^s consists of bit values from three fields – *source IP, source port*, and *protocol* of the flow rule. The destination address P^d contains bit values from the remaining two fields – *destination IP* and *destination port* of the flow rule. Using this organization of header fields, we define following spaces for representing a flow path space:

1. *Incoming Space* (S_i^P): It represents *original* header spaces of packets that can pass through the flow path, denoted as $[P_i^s, P_i^d]$.
2. *Outgoing Space* (S_o^P): It represents *final* header spaces of packets after the packets pass through the flow path, denoted as $[P_o^s, P_o^d]$.
3. *Tracked Space* (S_t^P): This space represents *original source* address and *final destination* address of header spaces of packets that can pass through the flow path. Thus, it is a combination of the source address of the incoming space (P_i^s) and the destination address of outgoing space (P_o^d), denoted as $[P_i^s, P_o^d]$.

Fig. 4(a) depicts the relationships of three kinds of flow path spaces. The *incoming space* of a flow path is calculated from the header spaces of incoming packets of the flow. The *outgoing space* of the flow path is computed from the header spaces of outgoing packets of the flow. Ultimately, the *tracked space* of the flow path is then derived from the source address of the incoming space and the destination address of the outgoing space. An example is given in Fig. 4(b), which illustrates the space representation of a *wildcard shifted* flow path. The incoming space of this flow path, [(*A, B*), (*C, D*)], indicates that *Host A* and *Host B* can send packets to *Host C* and *Host D* through this flow path, while the outgoing space of the flow path, [(*E, F*), (*M, N*)], presents that *Host E* and *Host F* can send packets to *Host M* and *Host N*. That is, any packets sent from *Host A* and *Host B* through this flow path will be delivered to *Host M* or *Host N*. Thus, the tracked space of this path is composed of the source address from its incoming space and the destination address from its outgoing space, represented as [(*A, B*), (*M, N*)].

To calculate the flow path spaces, we use a propagation function as described in Algorithm 1, which checks for the reachability of this packet from a source to a target destination. We leverage real time Header Space Analysis (Kazemian et al., 2013) (HSA) to compare header fields of a sample packet being propagated in SFPG against the match fields present in the flow rules. HSA is also used implicitly in algorithm to calculate the overlaps of header spaces (required for wildcarded and masked addresses). After a successful match, the corresponding actions present in the matched flow rules are applied on the sample packet.

Algorithm 1: Packet propagation in *SFPG*.

Input: Shifted Flow Path Graph, G_s; source node, s; target node, t; sample packet, pkt
Output: Tracked flow path space, S_v

```
1  /*Base case for the recursion*/
2  if pkt.node = target then
3      reached ⟵ true;
4      break;
5  /*Match the packet against flow rules in present switch*/
6  foreach rule ∈ pkt.node.getRules() do
7      /* Match Layer-2,3,4 fields of packet and flow rule */
8      if HSA.equals(pkt(l2, l3, l4), rule(l2, l3, l4)) then
9          /*Apply action(s) from the matched flow rule*/
10         foreach action ∈ rule.getActionSet() do
11             if action = set − field then
12                 pkt = action.applySetfield(pkt);
13                 continue;
14             else if action = forward then
15                 /*Propagate packet from the new node*/s = action.getNode();
16                 pkt.presentNode = s;
17                 propagate(G_s, s, t, pkt);
18                 /*Update the tracked space*/
19                 S_v = S_v + pkt(l2, l3, l4);
20                 if reached then
21                     return;
22             else if action = deny then
23                 return;
24 return S_v;
```

OpenFlow allows multiple actions including modification of header fields, forwarding the packet, and dropping the packet. A flow rule can also contain more than one action to be taken on a matched packet. Algorithm 1 applies different actions to the packet being propagated. It uses *applySetfield()* method for modifying header fields as per the action. Upon receiving a *forward* action, *propagate* function is called again with the new source node address set as the next hop (as mentioned in the flow rule). We maintain the history of packet throughout traversal by storing the path that the packet takes during the course. If the packet reaches target, the algorithm stops and returns the tracked space of the packet.

4.1.3. Firewall authorization space partition

In many cases, a system administrator may intentionally introduce certain overlaps in firewall rules by assuming the implicit priority of these rules (as used by in Linux *iptables*). In reality, this is

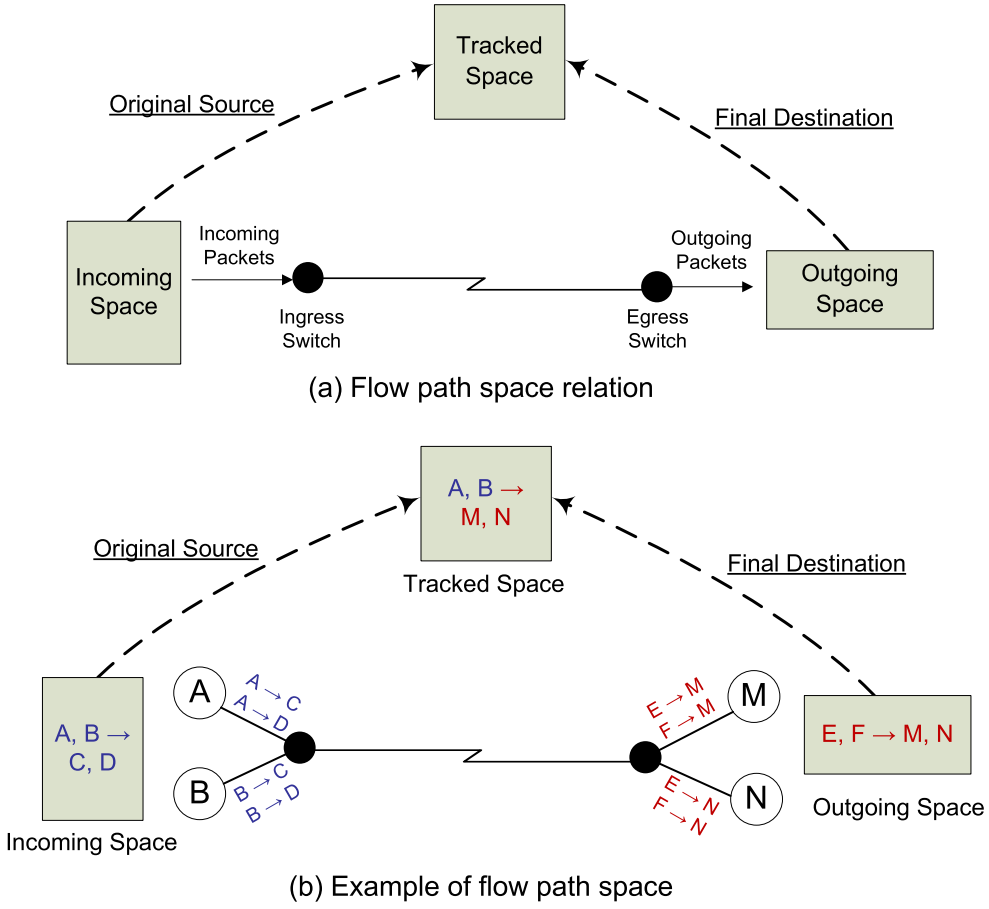


Fig. 4. Flow path space classification.

a commonly used technique to exclude specific parts from a certain action, and the proper use of this technique could result in a fewer number of *compact* rules (Yuan et al., 2006). Hence, for the purpose of accurately detecting firewall policy violations in OpenFlow networks, the dependency relations between "allow" rules and "deny" rules in the firewall policy should be decoupled.

We first introduce a concept of *Firewall Authorization Space*, which represents a collection of all packets either allowed or denied by the firewall rules. Then, we represent rules with *header space* and perform various set operations on rules to convert a list of firewall rules into two *disjoint* authorization subspaces – *denied authorization space* and *allowed authorization space*. Algorithm 2 shows the pseudocode of partitioning authorization space for a set of firewall rules R. The algorithm sequentially examines a header space s_r derived from a rule r and adds it to corresponding firewall authorization space sets, S_a^F or S_d^F, based on its type. For each r in R, if this rule is an "allow" rule, the header space s_r derived from this rule is compared with existing header spaces in the denied space set S_d^F. If the header space s_r is covered by any existing header spaces in S_d^F, the covered space(s) is removed from s_r and then the modified s_r is added into S_a^F. The similar process is applied to a "deny" rule. In this way, we can utilize set operations to separate the overlapped spaces of a firewall policy into two disjoint authorization space sets S_a^F: $\{s_{a_1}^F, ..., s_{a_{n-1}}^F, s_{a_n}^F\}$ and S_d^F: $\{s_{d_1}^F, ..., s_{d_{m-1}}^F, s_{d_m}^F\}$. Formally, $s_{a_i}^F \cap s_{d_j}^F = \varnothing$, where $s_{a_i}^F \in S_a^F$, $s_{d_j}^F \in S_d^F$, $1 \le i \le n$, and $1 \le j \le m$. Note that it is unnecessary to eliminate overlapping header spaces within S_a^F and S_d^F, since those overlapping header spaces could not affect the results of violation detection and keeping them can potentially reduce the number of header spaces in each authorization space set.

Algorithm 2: Partitioning firewall authorization space.

Input: A set of rules, R.
Output: A set of allowed spaces, S_a^F; A set of denied spaces, S_d^F.

```
1  foreach r ∈ R do
2      s_r ⟵ HeaderSpace(r);
3      if Action(r) = allow then
4          foreach s ∈ S_d^F do
5              /* s_r is overlapping with s*/
6              s_r ⟵ s_r \ s;
7              S_a^F.Append(s_r);
8      if Action(r) = deny then
9          foreach s' ∈ S_a^F do
10             /* s_r is overlapping with s'*/
11             s_r ⟵ s_r \ s';
12             S_d^F.Append(s_r);
13 return S_a^F, S_d^F;
```

An example of firewall authorization space partition is shown in Fig. 5. For the purposes of brevity and understandability, we employ a two-dimensional geometric representation for each header space derived from firewall rules. Note that a firewall rule typically utilizes five fields to define the rule condition, thus a complete representation of header space should be multi-dimensional. In Fig. 5(b), we utilize colored rectangles to denote two kinds of authorization spaces: *allowed space* (in white) and *denied space* (in pink), respectively. In this example, there is an allowed space representing the first rule and a denied space depicting the second rule. Two spaces overlap when there are packets matching both rules (Fig. 5(b)). Applying Algorithm 2 to the example policy (Fig. 5(a)), the header space of the first rule is added into the

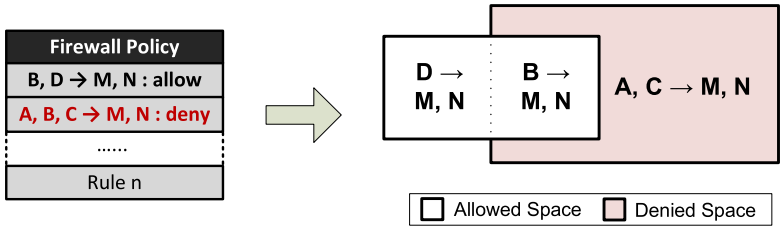


Fig. 5. Example of firewall authorization space.

allowed authorization space set. Then, the overlapped space is removed from the header space of the second rule, and the modified header space is added to the denied authorization space set.

4.1.4. Violation discovery

We use Algorithm 3 to identify violations by checking the tracked space (S_t^P) of a flow path against the firewall denied authorization space (S_d^F). For every rule in the denied authorization space, we store the source and destination nodes by using the *findDpid()* method which returns the combination of switch and node connector for the corresponding hosts. A sample packet is created by fetching fields corresponding to the IP 5-tuple set from the firewall rule.

If the algorithm detects an overlap between the tracked space and the firewall denied authorization space, we store and return the overlapping space. We call the returned space as violated space, $S_v = S_t^P \cap S_d^F$, denoted by $[P_v^s, P_v^d]$, where s and d denote source and destination addresses, respectively. Depending on the complexity of an overlap found in violated space, we categorize the violations as:

- *Entire Violation:* If the denied authorization space S_d^F includes the whole tracked space S_t^P of the flow path, the violated space S_v indicates an entire violation. Formally, $S_t^P \subseteq S_d^F$.
- *Partial Violation:* If the denied authorization space S_d^F partially includes the tracked space S_t^P of the flow path, the violated space S_v points out a partial violation. Formally, $S_t^P \not\subseteq S_d^F$ and $S_t^P \cap S_d^F \neq \varnothing$.

Algorithm 3: Violation detection.

Input: Denied authorization space, S_d^F; Shifted Flow Path Graph, G_s.
Output: A set of violating spaces, V;

```
/*Iterate through firewall rules in denied authorization space*/
foreach r ∈ S_d^F do
    source = findDpid(r.src);
    target = findDpid(r.dst);
    /*Create sample packet with header space from the deny rule*/
    packet = {r.src, r.dst};
    /*Propagate the packet and get the tracked space*/
    S_t^P = propagate(G_s, source, target, packet);
    S_v = S_t^P ∩ S_d^F;
    if S_v then
        V.Append(S_v)
return V
```

Fig. 6 shows an example of our violation detection approach. The tracked space depicts that the original source of the flow is *Host A* and *Host B*, and the final destination of the flow is *Host M* and *Host N*. The firewall authorization space illustrates that all packets from *Host A* and *Host C* to *Host M* and *Host N* are denied. Thus, the violated space, which is depicted in inclined stripes in

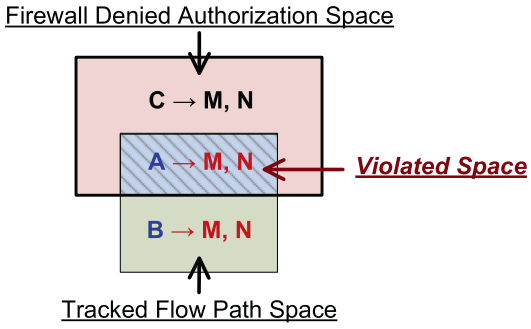


Fig. 6. Violation detection.

Fig. 6, contains a partial tracked flow space represented with the original source of *Host A* and the final destinations of *Host M* and *Host N*. That is, all packets originally sent from *Host A* and finally arrived at *Host M* or *Host N* should be denied by the firewall.

4.2. Violation resolution

An intuition for resolving a firewall policy violation is to simply disable the violated flow policy. That is, for a new flow policy, the request for installing this policy is rejected if the firewall application detects this policy is in violation of the firewall policy. Regarding the existing flow policies that violate the firewall policy, they are removed from the network devices directly. However, such a solution has several drawbacks. First, a flow policy may only *partially* violate the firewall policy as we discussed in Section 4.1.4. In this case, rejecting/removing the flow policy may affect the utility of network services. Second, a rule in a flow policy may have dependency relations with the rules of other flow policies. Deleting the rule in a violated policy may impact other flow policies and even create new violation(s). Therefore, it is necessary to seek a systematic solution to enable a flexible and effective violation resolution. To this end, we introduce a violation resolution framework, as depicted in Fig. 7, which demonstrates how FlowMon adopts four violation resolution strategies to resolve different firewall policy violations.

4.2.1. Dependency breaking

Situation: A new flow policy is being added to the network switches and no flow policy violates the firewall policy. However, the rules in this new flow policy overlap with the rules of other flow policies in the flow tables. In addition, as explained the Scenario 2 demonstrated in Section 3.2, these rule dependencies could cause new firewall policy violation(s). This kind of violation can also be incurred by other changes of network states, such as modifying flow entries and updating firewall rules.

Solution: A resolution approach for the issue is to break the dependencies among flow policies. Then, we can guarantee that

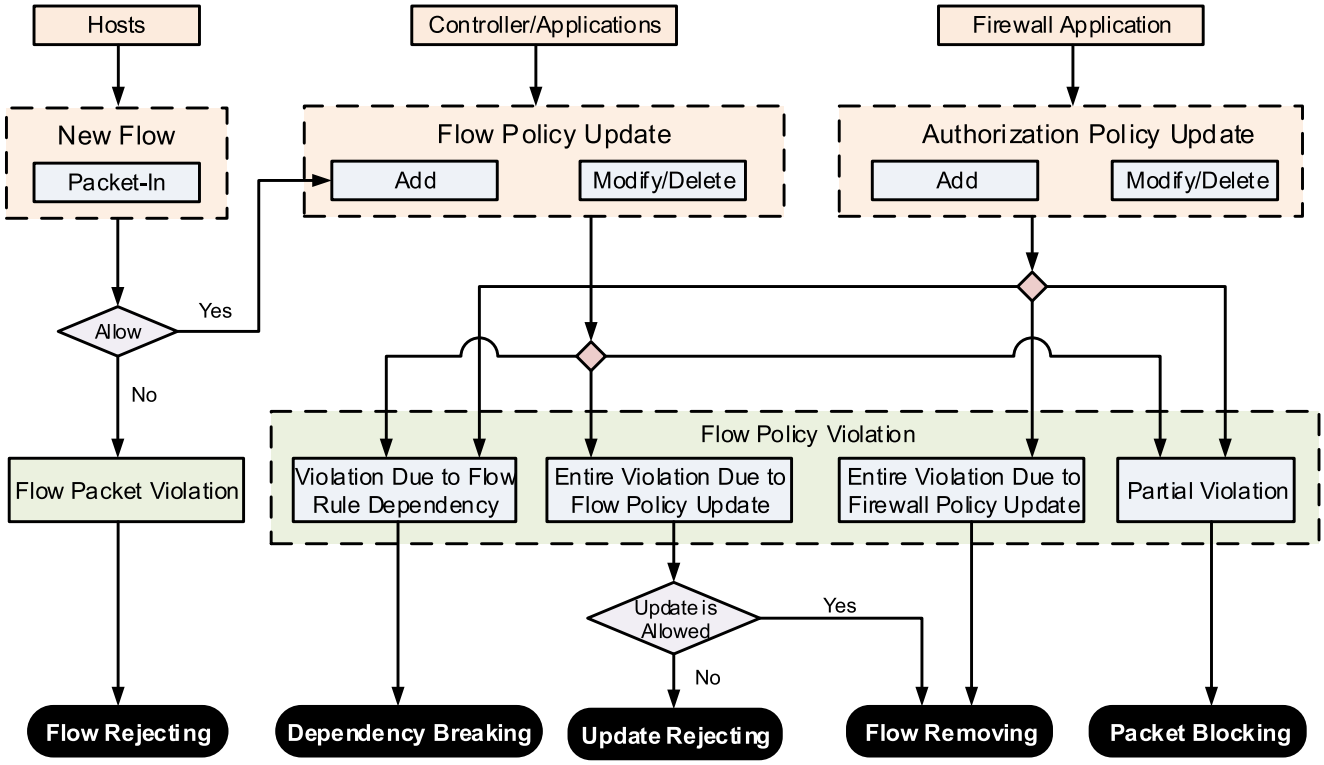


Fig. 7. FlowMon violation resolution framework.

when the packets of a flow traverse the network, they are precisely processed as defined by the policy for such a flow.

The approach introduced in Reitblatt et al. (2012) utilizes tags to distinguish packets belonging to different policies for ensuring consistent network updates. Inspired by the approach, we use *flow tagging* mechanism to break the rule dependencies in our violation resolution framework. In this mechanism, the new flow policy is preprocessed by adding a tag to differentiate the match pattern with other policies. The rule of the policy in the ingress switch will take additional action on the packets to stamp them with the same tag. In other words, each flow rule that causes rule dependency can enforce tagging to the corresponding flows, for example, using VLAN tagging to distinguish them from the packets managed by other flow rules. Similarly, flow rules managing the same flow stored in intermediate switches (as illustrated in Fig. 2) can check if each packet is tagged. If the packet is not tagged, the action corresponding to the packet can be carried out only after the packet is tagged. In this way, flows managed by different rules that may cause the rule dependency can be differentiated from each other and processed by the corresponding flow rules only. As the packets leave the network through egress switch, the corresponding rule of the policy will strip the tag off the packets.

4.2.2. Update rejecting

Situation: There are three possible cases in flow rule update that can apply this strategy: (1) when adding a new flow policy, corresponding flow path is detected as a violation of the firewall policy and the violation is an *entire* violation; (2) changing a flow rule induces new *entire* violation(s); and (3) deleting a flow rule causes new *entire* violation(s), since some rules of other flows have dependency relations with this flow rule.

Solution: The update operation is rejected directly to resolve the violation(s). Note that, this strategy may not be always applied for cases (2) and (3), since a change or delete operation on a rule may be mandatory depending on the privileges of the operator. [4]

4.2.3. Flow removing

Situation: There are two cases: (1) when updating (adding, changing, or deleting) a rule(s) in the firewall policy, the firewall examines the current network state applying the updated firewall rule(s) and detect new entire violation(s); and (2) a change or delete operation on a flow rule is allowed, even though it causes entire violation(s).

Solution: All rules associated with a flow path, which entirely violates the firewall policy, are removed from the network switches.

4.2.4. Packet blocking

Situation: For any *partial* violation detected by the firewall application, this strategy can be applied.

Solution: There may exist two ways to block packets of a flow: (1) if the flow is a new one, the firewall application only needs to block it in the ingress switch of the flow; and (2) if the flow is an old flow, the firewall application blocks the packets in both ingress and egress switches. In such a case, blocking packets at the ingress switch can prevent any new packets of the violated flow entering the network, while blocking packets at the egress switch can prevent any *inflight* packets of the violated flow from going through the network.

In order to block packets, the firewall application installs new blocking rules at the ingress and/or egress switches. As shown in Fig. 8, the blocking rules can be derived from the violated space (S_v: $[P_v^s, P_v^d]$). Header space of the blocking rule for the ingress switch is a combination of the source address of the violated space

[4] A permission system for OpenFlow controller similar to the one discussed in Wen et al. (2013) is required to decide the operator's privileges.

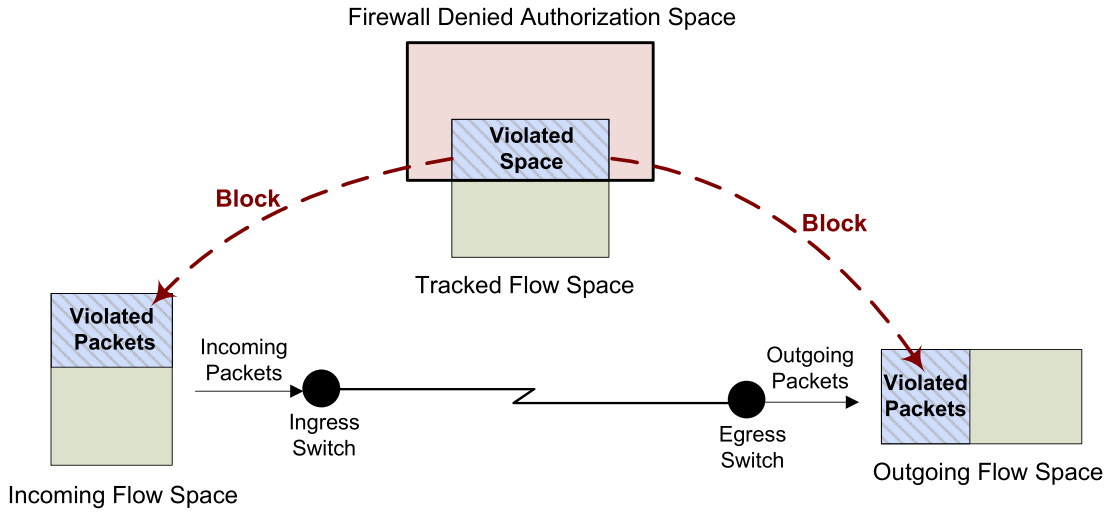


Fig. 8. Violation resolution through packet blocking.

(S_v) and the destination address of the incoming space (S_i^p), denoted as $[P_v^s, P_i^d]$. Header space of the blocking rule for the egress switch is combined from the source address of the outgoing space (S_o^p) and the destination address of the violated space (S_v), denoted as $[P_o^s, P_v^d]$.

4.3. Optimization considerations

Since an OpenFlow firewall application must perform the violation detection and resolution in real time, several optimization mechanisms should be considered. In this section, we introduce two approaches that can further improve the performance of FlowMon. Note that the following approaches are currently not implemented in our FlowMon and will be explored in depth in the future.

Incremental Checking: Built on NetPlumber, the flow track mechanism in FlowMon is capable of performing incremental checks when updating *flow policies*. Similarly, the concept of incremental *firewall policy* checks can be applied to FlowMon to increase its performance. Instead of recomputing the entire firewall authorization space whenever the firewall policy changes, FlowMon can only incrementally calculate the header spaces that are affected by these changes.

Maintaining Partial Flow Graph: As we explain in Section 4.1.2, FlowMon only needs to maintain an SFPG, which is a sub-graph of the plumbing graph. In addition, FlowMon can check the source address in the incoming space of each shifted flow path against the source address of head spaces in its denied space. If these two source addresses do not overlap each other, FlowMon can guarantee that the shifted flow path will not violate its policy without tracking the flow path. In this way, shifted flow path can be removed from SFPG. However, this approach is only applicable to detecting direct violation and cannot detect indirect violations caused by rule dependencies. Thus, shifted flow path should be utilized for detecting other types of violations.

5. Implementation and evaluation

We implement FlowMon[5] on top of Floodlight to demonstrate its functionality, performance and scalability. We also demonstrate that FlowMon can satisfy our design requirements (i.e., accuracy, flexibility, and efficiency) for OpenFlow-based firewall application. FlowMon adopts NetPlumber data structure (Header Space Library) for building header objects and computing intra-table dependencies. To that end, FlowMon contains several classes, such as *HeaderObject* and *RuleNode*. *HeaderObject* collects L2 ~ L4 flow headers to enable bit-level representation of packet headers, while *RuleNode* collects flow rules for checking rule dependency. For retrieving network topology information, we implement listeners to monitor the modules responsible for collecting such information. For example, *Static Flow Pusher* and *Memory Storage Source* are the two modules provided by Floodlight controller to retrieve network topology information and flow rules in real time. Our implementation of FlowMon retrieves flow rules using the *Static Flow Pusher* module of the controller and build/modify *RuleNodes*. In this way, FlowMon can sort the rules by priorities and compute intra-table dependencies. FlowMon also obtains the information of network devices including attached switch ID and corresponding port number using the *Memory Storage Source* module, and utilizes these information to understand the physical topology of the network.

FlowMon combines the collected topology information and flow rules installed in the network to build the flow graph for tracking flow paths. If the tracked spaces of flow paths overlap with any denied authorization spaces of a firewall policy, FlowMon analyzes the root cause of each violation and corresponding resolution strategy is applied to resolve the identified violation as illustrated in Fig. 7. At the same time, FlowMon maintains updated flow rules and network topology information so that it is able to not only re-propagate header objects at any associated switches to update flow paths, but also track and manipulate associated flow rules regarding *shifted flows*. In addition, FlowMon utilizes the Floodlight built-in firewall to generate new blocking rules and the *Static Flow Pusher* module to add/modify/delete flow rules for resolving violations with respect to different violation resolution strategies.

5.1. Experiment design

To demonstrate functionality, performance, and scalability of FlowMon we conduct our experiments in two different network topologies. One of the experiment network topology is a simple one as depicted in Fig. 2 to clearly demonstrate the functionality of FlowMon, which includes violation detection and resolution. For the other topology we used a *real-world* network (Stanford back-

[5] The source code for FlowMon is available for download at https://github.com/shishkebab/flowguard.

Table 1
Detection and resolution elapsed time in ms for different resolution strategies.

Resolution Strategy	Example Topology		Stanford Topology without Flow Entries		Stanford Topology with Flow Entries	
	Detection	Resolution	Detection	Resolution	Detection	Resolution
Dependency Breaking	1.70	2.91	3.97	3.98	4.58	4.34
Update Rejecting	1.83	2.45	3.46	3.57	4.83	3.73
Flow Removing	1.85	2.17	3.38	3.22	4.39	3.71
Packet Blocking	1.91	1.33	5.05	2.22	6.48	2.53

bone network) to show the scalability and performance of FlowMon. Stanford backbone network Kazemian et al. (2013) consists of 14 operational zone Cisco routers, 10 Ethernet switches, and 2 backbone Cisco routers. All of our experiments were performed in Ubuntu 12.04 virtual machines, each of which has four processors and 8GB memory. We run Mininet ver.2.0 (Mininet) in one virtual machine to simulate the network topologies, while another virtual machine runs FlowMon on top of Floodlight v0.90.

The entire configuration of the Stanford backbone network was retrieved from public header space library (Header Space Library). We parse the collected Standford dataset and Cisco access control list (ACL) files so that they can be used as an input to Floodlight. Using the input, corresponding firewall rules are generated for FlowMon. Total number of 8908 flow entries and 1206 *real* firewall rules are used in our experiments.

5.2. FlowMon violation detection and resolution

Dependency Breaking: First, we simulate dependency breaking casued by *shifted flow paths* (as explained in *Bypass Scenario 2* in Section 3.2) to demonstrate the detection functionality of FlowMon. We also show the effectiveness and performance of FlowMon in Stanford backbone network by measuring detection and resolution time in milliseconds (ms).

For the first part of the experiment (simulating *Bypass Scenario 2*), initial flow table entries for each switch are added as illustrated in Fig. 9. There was only one firewall rule specified in FlowMon, which denied the communications from 10.0.0.1 to 10.0.0.3. As explained in Section 4, *Flow 1* and *Flow 2* in Fig. 2 does not violate the firewall rule directly. However, the intra-table dependency in Switch 2 causes a potential firewall rule violation. In order to resolve this issue, FlowMon removes the intra-table dependency by isolating *Flow 1* from *Flow 2* using the VLAN field in flow entries to set up tags. The action field of the first flow entry in Switch 1 is updated to add a random VLAN ID 100, which is decided by FlowMon, and the corresponding flow entry in Switch 2 is modified to match the same VLAN ID. And finally in Switch 3, which is the egress switch in this case, the VLAN IDs are removed to restore the original packet. The flow tables of Switches 1, 2, and 3 before applying the resolution technique are shown in Figs. 9(a), (c), and (e). After finding the violation, the flow tables are updated by FlowMon. The updated table entries are shown in Figs. 9(b), (d), and (f), respectively.

We measure the violation detection and resolution time taken by FlowMon. The time taken for FlowMon to detect the dependency breaking in the experiment explained above is 1.70 ms, while the resolution time to resolve the detected issue is 2.91 ms (see Table 1). We also carry out the same experiment twice in the Stanford backbone network, one without any flow entries in the network, and the other with flow entries mentioned in Section 5.1 (i.e., 8908 flow entries and 1206 firewall rules). The detection and resolution time taken in Stanford topology slightly increases – 3.97 ms for detection and 3.98 ms for resolution *without* flow entries, and 4.58 ms for detection and 4.34 ms *with* flow entries (see Table 1). The detection and resolution time increases with flow entries and firewall rules installed in the network, because FlowMon must go through every flow entry and firewall rule to detect violation and isolate related flows and resolve it. Although both detection and resolution time increases by running FlowMon in Stanford backbone network, it is negligible overhead given the fact that it is run in *real-world* network with realistic number of flow entries and firewall rules.

Update Rejecting: We use the scenario explained in *Bypass Scenario 1* (Section 3.2) to test update rejecting strategy. We first install the flow entries in Switch 1 and Switch 3 as illustrated in Fig. 1. Then, we add two additional flow entries to Switch 2 that cause firewall rule violation. FlowMon not only detects the violation successfully but also decides to reject the flow table update request (see Fig. 10). FlowMon takes 1.83 ms to detect this violation and 2.45 ms for resolution in the example Mininet topology. In the Stanford network, FlowMon takes 3.46 ms for detection and 3.57 ms for resolution *without* the flow entries and firewall rules, while 4.83 ms for detection and 3.73 ms for resolution are taken *with* the flow entries and firewall rules (See Table 1).

Flow Removing: We utilize *Bypass Scenario 1* explained in Section 3.2 again to evaluate this strategy. We first set up all the flow entries as shown in Fig. 1. Then, we enable FlowMon and specify the firewall rule to deny packets from the sources A to C (Fig. 1). FlowMon successfully detects the violation and remove corresponding flow entries that cause this violation in the three switches, while the built-in Floodlight firewall cannot identify any violation. Similar to previous experimental results, FlowMon takes 1.85 ms to detect the violation and 2.17 ms to resolve it in our example topology. In the Stanford network topology *without* any flow entries, FlowMon takes 3.38 ms to detect the violation and 3.22 ms to resolve it. In the same topology *with* the flow entries, FlowMon takes 4.39 ms to detect the violation and 3.71 ms to resolve it.

Packet Blocking: To set up a partial violation, we first installed a flow policy so that the flow entry in Switch 1 forwards every packet whose source IP is in 10.0.0.0/24 and destination IP is in 10.0.0.0/24. The flow entry in Switch 2 modifies source IP to 10.0.0.6/32 when the incoming packet has source IP 10.0.0.1/32 and destination IP in 10.0.0.0/24. The flow entry in Switch 3 rewrites destination IP to 10.0.0.4/32 whenever the source IP is in 10.0.0.0/24 and the destination IP is 10.0.0.5/32 (See Fig. 11). Then, we installed a firewall rule that blocks packets from 10.0.0.1 to 10.0.0.4 to create a violation. The built-in Floodlight firewall cannot detect this violation. However, FlowMon detects the partial violation, generates two additional flow entries, and installs them in the ingress switch and egress switch, respectively.

As shown in Fig. 11(b), FlowMon installs a new flow entry in Switch 1 that blocks packets with source IP 10.0.0.1/32 and destination IP 10.0.0.5/32. Since the flow entry in Switch 2 rewrites the source IP of packets from 10.0.0.1/32 to 10.0.0.6/32, FlowMon also installs a new flow entry in Switch 3 that drops packets with source IP 10.0.0.6/32 and destination IP 10.0.0.5/32 as shown in Fig. 11(b). In addition, FlowMon installs a new firewall rule that denies the packets from 10.0.0.1 to 10.0.0.5. FlowMon

Priority	Match	Action
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 3
10	port=2, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.4	src=167772161, output 3

(a) Switch 1 Flow Table before Resolution

Priority	Match	Action
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 3, VLAN=100
10	port=2, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.4	src=167772161, output 3

(b) Switch 1 Flow Table after Resolution

Priority	Match	Action
10	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	src=167772162, dest=167772163, output 2
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2

(c) Switch 2 Flow Table before Resolution

Priority	Match	Action
1	port=1, VLAN=100, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2
10	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	src=167772162, dest=167772163, output 2

(d) Switch 2 Flow Table after Resolution

Priority	Match	Action
10	port=3, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.3	output 1
1	port=3, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2

(e) Switch 3 Flow Table before Resolution

Priority	Match	Action
10	port=3, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.3	output 1
1	port=3, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2, strip VLAN

(f) Switch 3 Flow Table after Resolution

Fig. 9. Flow tables before/after FlowMon's dependency breaking strategy. Every subfigure is a screenshot taken from the Floodlight controller GUI.

takes 1.90 ms for detection and 1.33 ms for resolution in the example Mininet topology. In the Standford network *without* flow entries, it takes 5.05 ms and 2.22 ms for detection and resolution respectively, and 6.48 ms and 2.53 ms *with* the flow entries. Our observation shows that the detection time imposes slightly more overhead than other results. This is because tracking flow paths for identifying partial violation requires additional steps involving both ingress and egress switches. Thus, the detection relatively takes more time than detecting other types of violation.

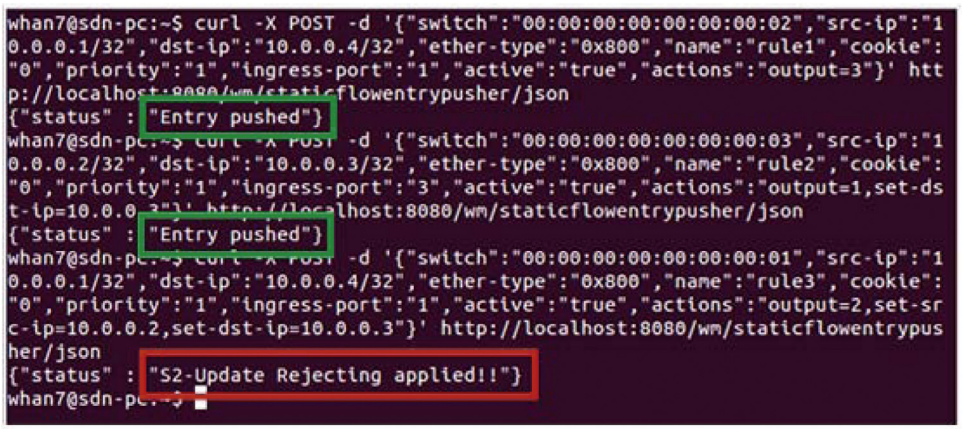


Fig. 10. Update rejecting result.

Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.0	output 3

(a) Switch 1 Flow Table before Resolution

Priority	Match	Action
32767	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.5	
0	port=1, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.0	output 3

(b) Switch 1 Flow Table after Resolution

Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.0	src=167772166, output 2

(c) Switch 2 Flow Table before Resolution

Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.0	src=167772166, output 2

(d) Switch 2 Flow Table after Resolution

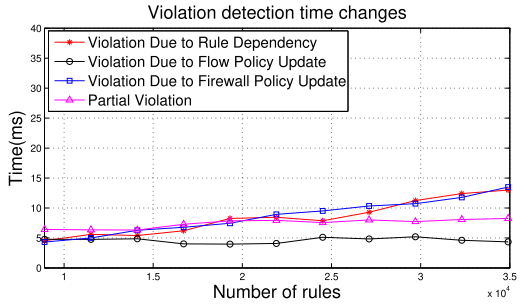
Priority	Match	Action
0	port=3, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.5	dest=167772164, output 2

(e) Switch 3 Flow Table before Resolution

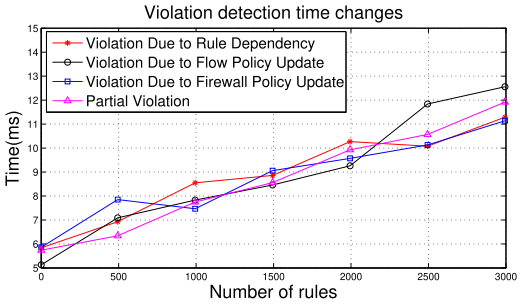
Priority	Match	Action
32767	port=3, ethertype=0x0800, src=10.0.0.6, dest=10.0.0.5	
0	port=3, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.5	dest=167772164, output 2

(f) Switch 3 Flow Table after Resolution

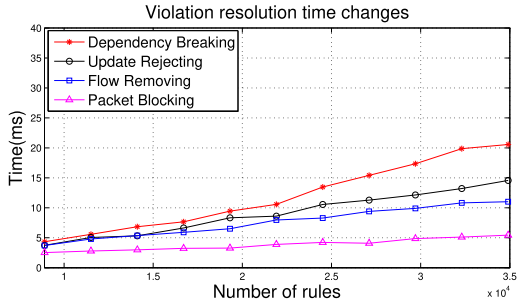
Fig. 11. Flow tables before/after FlowMon's packet blocking strategy. Every subfigure is a screenshot taken from the Floodlight controller GUI.



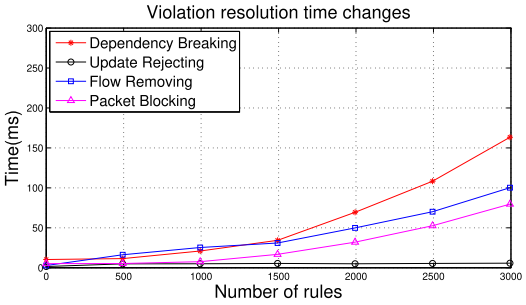
(a) Detection time changes in the first scenario



(b) Detection time changes in the second scenario



(c) Resolution time changes in the first scenario



(d) Resolution time changes in the second scenario

Fig. 12. Detection and resolution time changes under two different scenarios.

5.3. Scalability Analysis of FlowMon

We examine the scalability of FlowMon with respect to different sizes of flow rules in this Section. Especially, we aim to prove that our implementation of FlowMon satisfies *efficiency*, which is one of the design requirements for OpenFlow-based firewall application. To that end, we incrementally increases the number of flow rules in the Stanford network topology to evaluate the scalability of FlowMon. We observe detection and resolution time changes under two different test environments. In the first scenario, by inserting 100 $\sim$ 1000 additional flow rules in each switch, resulting in approximately 8.9 k $\sim$ 35 k rules in total in 26 switches in the Stanford network. In the second scenario, we increase the number of flow rules only in the switches associated with the violated flow paths. We incrementally add flow rules (500 $\sim$ 1500 rules at interval of 100 rules) at each relevant switch. Then, we install 8 firewall rules for each violation (i.e, violation due to rule dependency, flow policy update, firewall policy update, and partial violation) in FlowMon and measure the time taken by FlowMon to detect each violation. As shown in Fig. 12(a) and Fig. 12(b), the violation detection time was increased linearly in accordance with the growing number of flow rules. Note that the violation detection time decreases even if the number of rules increase in some cases in the first scenario shown in Fig. 12(a) due to a random distribution of rules in different switches. The resolution time changes in the second scenario, depicted in Figs. 12(d), indicate that dependency breaking strategy is the resolution mechanism that causes the highest overhead among the four resolution strategies, which is consistent with our previous experimental results (See Fig. 1). However, as illustrated in Figs. 12(c), FlowMon spends less than 25 ms to resolve each violation in the same network with larger numbers of flow rules. Even though all the rules installed in the first environment are not installed only in the switches associated with the violated flow paths, the number of rules is still considerable amount and indicate that FlowMon is efficient in terms of scalability.

5.4. Performance comparison with floodlight built-in firewall

In addition to demonstrating functionality and scalability analysis, we also compare the performance of FlowMon with that of Floodlight built-in firewall (FW). We first measure the time FlowMon and FW spend to initialize themselves in different network configurations. For an empty network where there is no network node, FlowMon takes 0.88 ms for initialization, while FW takes 0.87 ms. In the Stanford network *without* any flow entries, FlowMon takes 3.21 ms for initialization, while FW spends 1.02 ms. In the Stanford network *with* all forwarding entries and ACL rules installed, FlowMon takes 740.08 ms, while FW takes 0.97 ms. The reason why FlowMon takes significantly loner to initialize itself than FW is because FlowMon needs to analyze the entire flow entries, as well as firewall rules installed in the network to detect violation. However, the initialization speed of FlowMon can still be considered fast (less than one second), given the fact that it is run in a real-world network.

We also compare the time both FlowMon and FW take to update new firewall rules to observe if there is any significant performance overhead imposed by FlowMon. In the Stanford network, we update 700 random firewall rules to FlowMon and FW, and record the time each of the firewalls takes to process the rules. Using the output (i.e., the time taken to process the firewall rules updated) we plot an empirical cumulative distribution function (CDF) graph for each firewall not only to illustrate the results but also to show the probability distribution (Fig. 13). Overall, both FlowMon (denoted as FG in Fig. 13) and FW yield almost identical outputs, in terms of both actual time taken and probability distribution, with

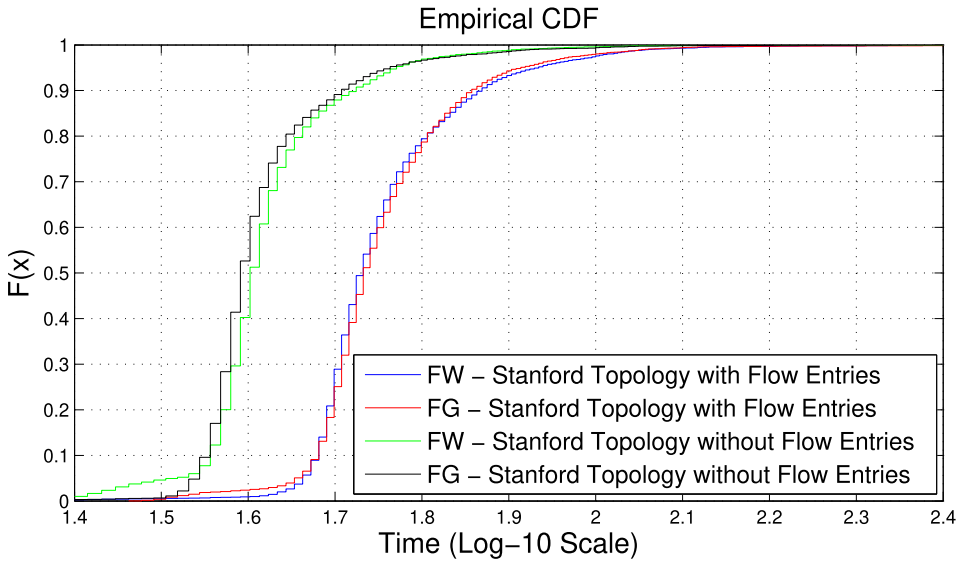


Fig. 13. Firewall rule update time in μs.

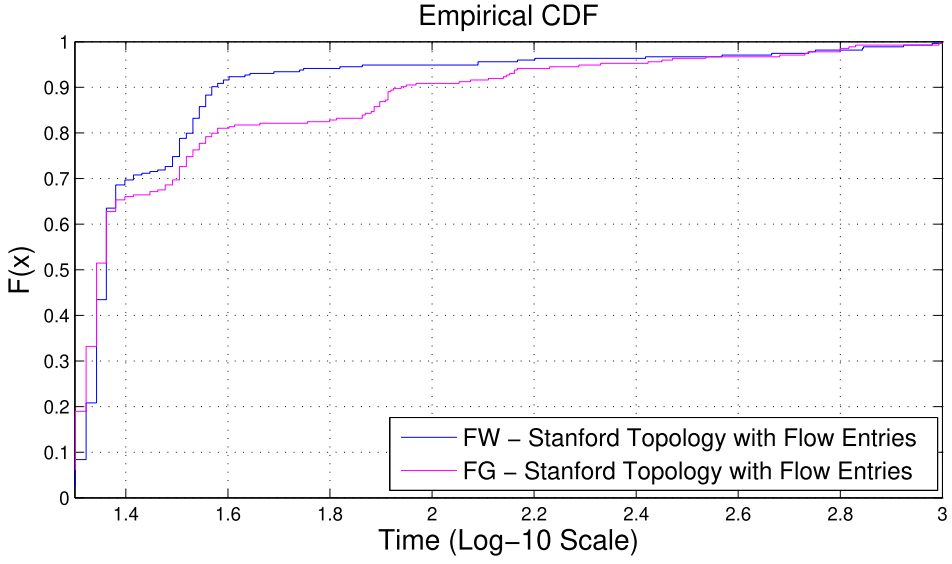


Fig. 14. Per packet inspection time in μs.

and without flow entries in the network. The rules updated to both firewalls are processed in less than 63 μs in the Stanford topology.

Finally, we generate 5000 test packets to measure per-packet inspection time of FlowMon and FW. We run the experiment in the Stanford topology with all firewall rules installed (i.e., 1206 rules) to match them against the test packets at per-packet level. Results show that the inspection time for 90% of test packets with flow entries takes 0.074 ms, while without flow entries FlowMon and FW spend less than 0.051 ms to inspect 90% of the packets. The results clearly indicate that FlowMon performs as fast as the built-in firewall, generating negligible overhead (Fig. 14).

6. Discussion

Security Enforcement Kernel: The design goal of FortNOX is to provide a security enforcement kernel (SEK) that can be integrated into OpenFlow controllers. Other OpenFlow-based security applications can rely on such an SEK to detect and resolve rule conflicts that may be introduced by non-security applications. FortNOX has been utilized to support FRESCO (Shin et al., 2013a), an OpenFlow security application development framework. However, FortNOX has several limitations in rule conflict detection and resolution. In contrast, FlowMon provides a new design that facilitates not only accurate conflict detection but also flexible and effective conflict resolution. Thus, we believe the solution provided by FlowMon could be potentially utilized for building a more robust SEK for OpenFlow controllers.

Stateful Monitoring: Currently, OpenFlow only provides very limited access to packet-level information in the controller (Shirali-Shahreza and Ganjali, 2013a; 2013b). In addition, the OpenFlow forwarding plane is almost stateless and incapable to actively monitor flow status without the involvement of the controller (Bianchi et al., 2014; Song, 2013). Therefore, as our first step for designing an OpenFlow-based firewall application, we only implement FlowMon as a stateless firewall application, which could not perform stateful packet inspection in OpenFlow networks. However, we also explore how FlowMon can be extended to support stateful packet inspection (Han et al., 2016; Hu et al., 2014a).

Flow Tracking: The flow tracking mechanism used in FlowMon is built on NetPlumber, which has limitations for dealing with middleboxes with dynamic state (Kazemian et al., 2013). FlowTags (Fayazbakhsh et al., 2013; 2014) is recently proposed to handle dy-

namic traffic modification in the presence of legacy middleboxes. However, FlowTags needs to extend current OpenFlow architecture to support flow tracking features. As our further work, we would like to study more effective flow tracking solution for FlowMon implementation.

Network Programming: The current design of FlowMon is built on top of OpenFlow, which is defined at a low level of network abstraction. PISCES (Shahbaz et al., 2016) is a derivative of software switches such as Open Vswitch (Pfaff et al., 2015), in an attempt to provide more flexibility for configuring how packets are processed by data plane elements. To that end, PISCES supports a protocol-independent programming language called P4 that enables the network administrators to re-program switches for custom packet processing. However, PISCES is specifically designed for the ease of network programming, and thus, lacks the support for security features such as flow tracking and detecting the divergence of flow paths.

The Frenetic Project (Frenetic) introduces a family of languages providing reusable and high level abstractions for programming SDNs. In particular, Pyretic (Monsanto et al., 2013), which is one member of the Frenetic family, enables a program to combine multiple policies together using policy composition operators to potentially resolve partial policy conflicts including *direct* firewall policy violations. However, lacking a policy conflict detection mechanism in Pyretic, it is obviously inefficient to *always* compose the firewall policy with flow policies and install them into the network switches due to several reasons. First, a firewall policy may consist of over thousands of rules, but commodity SDN switches with limited TCAM space typically support only a few thousands of rules (Stephens et al., 2012). Second, if flow policies *entirely* violates the firewall policy, it is unnecessary to install those violated flow policies into the network switches. Therefore, we would study for solutions that facilitate more secure and effective policy compositions in high level abstractions for building security applications in SDNs (Han et al., 2014).

7. Conclusion

In this paper, we have presented the design of a new OpenFlow-based firewall application, FlowMon, for software-defined networks. FlowMon provides an accurate and efficient approach to detect firewall policy violations through examining the flow path space against the firewall authorization space. In addition, FlowMon supports a flexible and fine-grained conflict resolution with respect to different update scenarios in flow entries and firewall rules. We have also implemented FlowMon to demonstrate its functionalities and performance, along with its capability to accurately detect the violations in real-time. Our experimental results show that FlowMon has the manageable performance overhead to enable real-time monitoring of software-defined networks.

As part of future work, we would like to study the optimization approaches discussed in Section 4.3 in depth and integrate those optimization approaches into FlowMon. We will also explore a more effective flow tracking solution for FlowMon and investigate how to integrate such a flow tracking mechanism into high-level SDN languages, such as Pyretic, enabling more secure SDN composition. Besides, we will investigate how the solution provided by FlowMon could be utilized to enhance existing security enforcement kernels (SEK), such as SE-Floodlight (SE-Floodlight), for OpenFlow controllers.

Declaration of Competing Interest

The authors declare no potential conflict of interest.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1642143, 1723663, 1700499, and 1642031.

References

Al-Shaer, E., Al-Haj, S., 2010. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration. ACM, pp. 37–44.
Al-Shaer, E., Hamed, H., 2004. Discovery of policy anomalies in distributed firewalls. In: Proceedings of the IEEE INFOCOM 2004 Conference, Vol. 4. IEEE, pp. 2605–2616.
Alfaro, J., Boulahia-Cuppens, N., Cuppens, F., 2008. Complete analysis of configuration rules to guarantee reliable network security policies. Int. J. Inf. Secur. 7 (2), 103–122.
Alshamrani, A., Chowdhary, A., Pisharody, S., Lu, D., Huang, D., 2017. A defense system for defeating DDoS attacks in SDN based networks. In: Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access. ACM, pp. 83–92.
Baboescu, F., Varghese, G., 2003. Fast and scalable conflict detection for packet classifiers. Comput. Netw. 42 (6), 717–735.
Benton, K., Camp, L.J., Small, C., 2013. OpenFlow vulnerability assessment (poster). In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13). ACM, pp. 151–152.
Bianchi, G., Bonola, M., Capone, A., Cascone, C., 2014. OpenState: programming platform-independent stateful OpenFlow applications inside the switch. ACM SIGCOMM Comput. Commun. Rev. 44 (2), 44–51.
Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S., 2007. Ethane: taking control of the enterprise. In: Proceedings of the ACM SIGCOMM 2007 Conference. ACM.
Casado, M., Garfinkel, T., Akella, A., Freedman, M.J., Boneh, D., McKeown, N., Shenker, S., 2006. SANE: a protection architecture for enterprise networks. In: Proceedings of the 15th Conference on USENIX Security Symposium. USENIX Association.
Fayazbakhsh, S., Sekar, V., Yu, M., Mogul, J., 2013. FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions. In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13).
Fayazbakhsh, S.K., Chiang, L., Sekar, V., Yu, M., Mogul, J.C., 2014. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.
Fundulaki, I., Marx, M., 2004. Specifying access control policies for XML documents with XPath. In: Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies. ACM, pp. 61–69.
Floodlight: Open SDN Controller. http://www.projectfloodlight.org.
Frenetic: A Family of Network Programming Languages. http://frenetic-lang.org/.
Greenberg, A., Hjalmtysson, G., Maltz, D.A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., Zhang, H., 2005. A clean slate 4d approach to network control and management. ACM SIGCOMM Comput. Commun. Rev. 35 (5), 41–54.
Han, W., Hu, H., Ahn, G.-J., 2014. LPM: layered policy management for software-defined networks. In: Proceedings of the 28th Annual WG 11.3 Conference on Data and Applications Security and Privacy (DBSec 2014).
Han, W., Hu, H., Zhao, Z., Doupé, A., Ahn, G.-J., Wang, K.-C., Deng, J., 2016. State-aware network access management for software-defined networks. In: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies. ACM, pp. 1–11.
Hari, A., Suri, S., Parulkar, G., 2000. Detecting and resolving packet filter conflicts. In: Proceedings of the IEEE INFOCOM 2000 Conference, Vol. 3. Citeseer, pp. 1203–1212.
Header Space Library. https://bitbucket.org/peymank/hassel-public.
Hong, S., Xu, L., Wang, H., Gu, G., 2015. Poisoning network visibility in software-defined networks: New attacks and countermeasures.. In: NDSS, Vol. 15, pp. 8–11.
Hu, H., Ahn, G.-J., Han, W., Zhao, Z., 2014. Towards a reliable SDN firewall. Open Networking Summit 2014 (ONS 2014) Research Track. USENIX.
Hu, H., Ahn, G.-J., Kulkarni, K., 2010. FAME: a firewall anomaly management environment. In: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration. ACM, pp. 17–26.
Hu, H., Ahn, G.-J., Kulkarni, K., 2012. Detecting and resolving firewall policy anomalies. IEEE Trans. Dependable Secure Comput. 9 (3), 318–331.
Hu, H., Han, W., Ahn, G.-J., Zhao, Z., 2014. FlowGuard: building robust firewalls for software-defined networks. In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14). ACM.
Ioannidis, S., Keromytis, A.D., Bellovin, S.M., Smith, J.M., 2000. Implementing a distributed firewall. In: Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00). ACM, pp. 190–199.
Jajodia, S., Samarati, P., Subrahmanian, V.S., 1997. A logical language for expressing authorizations. In: Proceedings of the IEEE Symposium on Security and Privacy. Oakland, CA, pp. 31–42.
Jantila, S., Chaipah, K., 2016. A security analysis of a hybrid mechanism to defend DDoS attacks in SDN. Procedia Computer Sci. 86, 437–440.

Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S., 2013. Real time network policy checking using header space analysis. In: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation. USENIX Association, pp. 99–112.

Kazemian, P., Varghese, G., McKeown, N., 2012. Header space analysis: static checking for networks. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX Association.

Khan, S., Gani, A., Wahab, A.W.A., Guizani, M., Khan, M.K., 2017. Topology discovery in software defined networks: threats, taxonomy, and state-of-the-art. IEEE Commun. Surv. Tutorials 19 (1), 303–324.

Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B., 2013. VeriFlow: verifying network-wide invariants in real time. In: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation. USENIX Association, pp. 15–28.

Kreutz, D., Ramos, F., Verissimo, P., 2013. Towards secure and dependable software-defined networks. In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13). ACM, pp. 55–60.

Lee, S., Yoon, C., Lee, C., Shin, S., Yegneswaran, V., Porras, P.A., 2017. Delta: a security assessment framework for software-defined networks. NDSS.

Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., Lin, D., 2009. Access control policy combining: theory meets practice. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies. ACM, pp. 135–144.

Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., King, S.T., 2011. Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference, pp. 290–301.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. 38 (2), 69–74.

Mininet: An Instant Virtual Network on Your Laptop. http://mininet.org.

Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., 2013. Composing software-defined networks. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, pp. 1–14.

Mousavi, S.M., St-Hilaire, M., 2015. Early detection of DDoS attacks against SDN controllers. In: 2015 International Conference on Computing, Networking and Communications (ICNC). IEEE, pp. 77–81.

OpenFlow Switch Specification. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al., 2015. The design and implementation of open vSwitch. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pp. 117–130.

Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G., 2012. A security enforcement kernel for OpenFlow networks. In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12).

Porras, P.A., Cheung, S., Fong, M.W., Skinner, K., Yegneswaran, V., 2015. Securing the software defined network control layer.. In: Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS).

Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D., 2012. Abstractions for network update. In: Proceedings of the ACM SIGCOMM 2012 Conference. ACM, pp. 323–334.

Sasan, Z., Salehi, M., 2017. SDN-based defending against ARP poisoning attack. J. Adv. Comput. Res. 8 (2), 95–102.

Schultz, E.E., 2002. A framework for understanding and predicting insider attacks. Comput. Secur. 21 (6), 526–531.

SE-Floodlight. http://www.openflowsec.org/.

Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., Rexford, J., 2016. PISCES: a programmable, protocol-independent software switch. In: Proceedings of the 2016 ACM SIGCOMM Conference. ACM, pp. 525–538.

Shin, S., Porras, P., Yegneswaran, V., Fong, M., Gu, G., Tyson, M., 2013. FRESCO: modular composable security services for software-defined networks. In: Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13).

Shin, S., Yegneswaran, V., Porras, P., Gu, G., 2013. AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks. In: Proceedings of the 20th ACM Conference on Computer and Communications security (CCS'13). ACM, pp. 413–424.

Shirali-Shahreza, S., Ganjali, Y., 2013. Efficient implementation of security applications in OpenFlow controller with FleXam. In: Proceedings of the IEEE 21st Annual Symposium on High-Performance Interconnects (HOTI),. IEEE, pp. 49–54.

Shirali-Shahreza, S., Ganjali, Y., 2013. FleXam: flexible sampling extension for monitoring and security applications in OpenFlow (poster). In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13).

Song, H., 2013. Protocol oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13).

Stephens, B., Cox, A., Felter, W., Dixon, C., Carter, J., 2012. PAST: scalable ethernet for data centers. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'12). ACM, pp. 49–60.

Wen, X., Chen, Y., Hu, C., Shi, C., Wang, Y., 2013. Towards a secure controller platform for OpenFlow application (poster). In: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13).

Yoon, C., Lee, S., Kang, H., Park, T., Shin, S., Yegneswaran, V., Porras, P., Gu, G., 2017. Flow wars: systemizing the attack surface and defenses in software-defined networks. IEEE/ACM Trans. Netw. (TON) 25 (6), 3514–3530.

Yuan, L., Chen, H., Mai, J., Chuah, C., Su, Z., Mohapatra, P., Davis, C., 2006. FIREMAN: a toolkit for firewall modeling and analysis. In: 2006 IEEE Symposium on Security and Privacy, p. 15.