FLOWGUARD: Building Robust Firewalls for Software-Defined Networks

Hongxin Hu[†], Wonkyu Han[‡], Gail-Joon Ahn[‡], and Ziming Zhao[‡]
[†]Clemson University [‡]Arizona State University
hongxih@clemson.edu, {whan7,gahn,zzhao30}@asu.edu

ABSTRACT

Software-Defined Networking (SDN) 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 robust firewalls for protecting OpenFlow-based networks where network states and traffic are frequently changed. To address this challenge, we introduce FLOWGUARD, a comprehensive framework, to facilitate not only accurate detection but also effective resolution of firewall policy violations in dynamic OpenFlow-based networks. FLOWGUARD checks network flow path spaces to detect firewall policy violations when network states are updated. In addition, FLOWGUARD conducts automatic and real-time violation resolutions with the help of several innovative resolution strategies designed for diverse network update situations. We also implement our framework and demonstrate the efficacy and efficiency of the proposed detection and resolution approaches in FLOWGUARD through experiments with a real-world network topology.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network monitoring; D.4.6 [**Security and Protection**]: Access controls

Keywords

Firewalls; Software-Defined Networking; OpenFlow; Security

1. INTRODUCTION

One primary goal of SDN is to enable a network controller to run various network services and manage the entire network directly by configuring packet-handling mechanisms in underlying devices. Consequently, enterprises adopt OpenFlow [16] for managing their networks in a cost-effective manner and at the same time, it is inevitable for their legacy security appliances such as firewalls and intrusion detection and prevention systems (IDS/IPS) to be migrated to OpenFlow-based networks by re-designing and implementing these systems as compatible security applications or services. In this paper, we focus on the challenges in designing and implementing *robust* firewalls for OpenFlow-based networks.

Firewalls are the most widely deployed security mechanism in most businesses and institutions. A conventional firewall sits on the border between a private network and the public 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 seen and cannot be filtered by the firewall [10]. That assumption has been invalid for a long time, because insiders could easily launch attacks on others in the network by circumventing security mechanisms [21]. With OpenFlow, such a problem could be potentially alleviated, since OpenFlow offers a deeper level of control granularity by placing enforcement points in any entries of traffic flows in a network. Our study reveals that OpenFlow not only presents tremendous opportunities to networking, but also brings great challenges for building SDN firewalls as follows:

- **Examining Dynamic Network Policy Updates:** 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 in a firewall application is not effective, since *flow policy violation*, which indicates existing flow policies violate the firewall policy induced by the proactive installation of flow policies, or the changes of network states and configurations such as updating flow entries and firewall rules, should be detected and resolved in real time as well.
- **Checking Indirect Security Violations:** OpenFlow allows various `Set-Field` actions that can dynamically change the packet headers. *Adversaries* could take advantage of this feature to strategically leverage flow rules that would evade network security mechanisms (e.g., firewalls) [19]. In addition, flow rules may overlap each other in a flow table, indicating intra-table dependency of flow rules [11]. The rules in a firewall policy may also overlap each other [26]. These rule dependencies could also be leveraged by *malicious* OpenFlow applications to bypass firewalls.
- **Architecture Option:** *A centralized SDN firewall*, which centrally defines and enforces the firewall policy on top of a controller, can immediately enforce updated rules in the firewall policy to check security violations. However, when a flow policy violation is detected, it can only reject the new flow policy or flush resident flow policy that causes the violation. OpenFlow allows wildcard rules in flow policies. Thus, if only *partial* packets matching a flow policy violate the firewall policy, eliminating the flow policy may drop legal traffic. In contrast, *a distributed SDN firewall* may directly

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620749.

resolve flow policy violations by propagating and enforcing the firewall policy at each *individual* entry (ingress switch) of the flow in the network. However, a distributed firewall needs a complicated revocation and repropagation mechanism [10] to handle dynamic policy updates.

- **Stateful Monitoring:** Currently, OpenFlow only provides very limited access to packet-level information in the controller [23]. In addition, the OpenFlow forwarding plane is almost stateless and unable to actively monitor flow status without the involvement of the controller [5]. Therefore, it is challenging to fully support stateful packet inspection in SDN firewalls.

In this work, we seek *a systematic solution for building robust firewalls that enable effective network-wide access control in emerging SDNs*. We first analyze and articulate several typical challenges in designing and implementing SDN firewalls, focusing on OpenFlow-based networks. To mitigate those challenges, we propose a comprehensive framework called FLOWGUARD that facilitates not only accurate detection but also flexible resolution of firewall policy violations in dynamic OpenFlow-based networks. The violation detection approach in FLOWGUARD detects violations by examining flow path spaces against the authorization space specified in the firewall, and is capable of tracking flow paths in the entire network and checking rule dependencies in both flow tables [11] and firewall policies [26]. Besides, FLOWGUARD determines violations dynamically when network states or configurations are changed. Especially, we introduce a flexible and effective violation resolution mechanism in FLOWGUARD to enable an automatic and real-time violation resolution, which has not been addressed by existing verification approaches for SDNs (e.g., [11, 13]), with the help of five resolution strategies, namely *flow rejecting*, *dependency breaking*, *update rejecting*, *flow removing*, and *packet blocking*, considering diverse update situations in networks. In order to effectively implement and deploy the proposed solution, we also integrate a variety of toolkits for supporting visualization, optimization, migration, and integration of SDN firewalls in FLOWGUARD.

This paper is organized as follows. We overview the related work in Section 2. Section 3 presents the FLOWGUARD framework in detail. We address the implementation and evaluation of FLOWGUARD in Section 4. We conclude this paper along with our future work in Section 5.

2. RELATED WORK

Several recent efforts have been devoted to address various security challenges, such as scanning attack prevention [17], DDoS attack detection [6], vulnerability assessment [14], and saturation attack mitigation [22], in SDNs. Differentiating from those work, our work focuses on exploring how to build robust firewalls for SDNs.

An exemplar SDN firewall application has been introduced in Floodlight [1] where each packet-in behavior triggered by the first packet of a traffic flow is matched against a set of existing firewall rules that allow or deny a flow at its ingress switch. Nevertheless, such a preliminary implementation of OpenFlow-based firewall application can only examine *flow packet violations* when *new* flows come in the network and cannot check *flow policy violations* with respect to dynamic network policy updates. Pyretic [18] was recently introduced as a higher-level language in the Frenetic Project [2] that allows SDN programmers to write modular network applications. Pyretic's sequential composition operator 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 [7]. FortNOX [19] was 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 FortNOX approach to design SDN firewalls by virtue of 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 network traffic 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 [11, 13] and among security constraints (represented as a firewall policy in our approach) [26]. 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. In [25], an earlier solution for building a security-enhanced firewall application was introduced. However, this solution only focuses on addressing *bypass* threats in OpenFlow-based networks. In contrast, FLOWGUARD is a comprehensive framework for building robust SDN firewalls to enable both accurate detection and flexible resolution of *various* firewall policy violations in dynamic OpenFlow-based networks.

A couple of verification tools [11, 12, 13, 15] for checking network invariants and policy correctness in OpenFlow networks have been proposed. Especially, VeriFlow [13] and NetPlumber [11] are capable of checking the compliance of network updates with specified invariants in real time. Even though these tools can be potentially used to detect firewall policy violations, they could not support automatic and effective violation resolution. Also, they ignore rule dependencies within security constraints, such as firewall policies, for compliance checking. In addition, they are unable to check stateful network properties [24].

Numerous firewall algorithms and tools have been designed to assist system administrators in managing and analyzing firewall policy anomalies [4, 8, 9, 26]. Yuan et al. [26] presented FIREMAN, a toolkit to check for misconfigurations in firewall policies through static analysis. Our previous work [8, 9] introduced FAME, a visualization-based firewall anomaly management environment, for detection and resolution of firewall anomalies. However, existing firewall policy analysis tools only detect policy anomalies *within* a firewall policy, but cannot be directly applied to deal with firewall policy violations against flow policies in dynamic OpenFlow networks with respect to network-wide access control.

3. FLOWGUARD **DESIGN**

In this work, our goal is to design a robust SDN firewall that supports network-wide access control by effectively managing firewall policy violations in dynamic OpenFlow-based networks. To achieve our goal and address the aforementioned challenges and limitations, we seek a solution that fulfills following design requirements.

1. *Accuracy.* The SDN firewall 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. [1]

[1]The detailed definitions are given in Section 3.1.3.

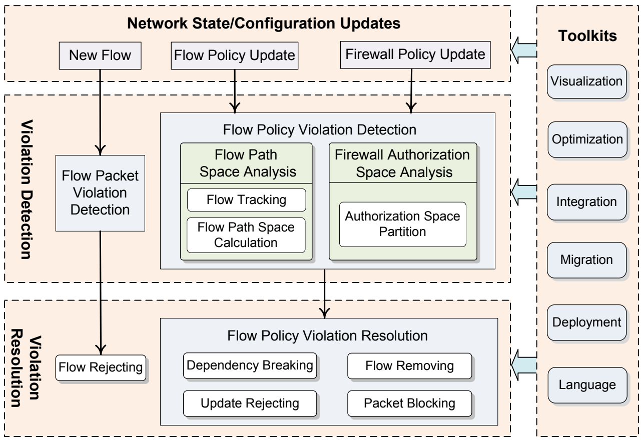


Figure 1: FlowGuard framework overview.

2. *Flexibility.* The SDN firewall 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 deal with fine-grained violation resolutions.
3. *Efficiency.* The SDN firewall needs to continuously work in a timely fashion. Also, the state of an OpenFlow-based network generally evolves rapidly. Thus, it naturally requires that the response time of an SDN firewall should be fast enough and its performance overhead should not affect other network utilities.

We propose a comprehensive framework, FlowGuard, to accommodate our design requirements. As shown in Figure 1, FlowGuard addresses several significant challenges in building SDN firewalls to facilitate accurate detection as well as flexible resolution of firewall policy violations in dynamic OpenFlow networks along with a variety of toolkits for visualization, optimization, migration, and integration of SDN firewalls. We next articulate the core components within the FlowGuard framework.

3.1 Violation Detection

Flow packet violations can be handled by using the traditional technique for firewall packet filtering. However, it is challenging to deal with flow policy violation, since both firewall and flow policies support *wildcard* rules. Moreover, in an OpenFlow network, the header fields of flow packets could be dynamically changed when the packets traverse the network. Thus, to support accurate violation detection and enable network-wide access control, a firewall application needs to not only check violations at the ingress switch of each flow, but also track the flow path and then clearly identify both the *original* source and *final* destination of each flow in the network.

3.1.1 Flow Path Space Analysis

Flow Tracking: To support network-wide access control in an OpenFlow network, a firewall application 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 [11, 12, 13] could check network reachability in real time and be potentially used to help find flow paths in OpenFlow networks. As a preliminary solution, we leverage Header Space Analysis (HSA) [11, 12] as a baseline for building the flow tracking mechanism in our framework, since it offers several features that can fulfill some of our design

Algorithm 1: Partitioning firewall authorization space

```
Input: A set of firewall rules, $R$.
Output: A set of allowed spaces, $S_a^F$; A set of denied spaces, $S_d^F$.
foreach $r \in R$ do
    $s_r \longleftarrow HeaderSpace(r)$;
    if $Action(r) = allow$ then
        foreach $s \in S_d^F$ do
            /* $s_r$ is overlapping with $s$ */
            $s_r \longleftarrow s_r \setminus s$;
        $S_a^F.Append(s_r)$;
    if $Action(r) = deny$ then
        foreach $s' \in S_a^F$ do
            /* $s_r$ is overlapping with $s'$ */
            $s_r \longleftarrow s_r \setminus s'$;
        $S_d^F.Append(s_r)$;
return $S_a^F$, $S_d^F$;
```

requirements for effective flow tracking: (1) it uses a geometric model (*header space*) of packet processing to provide a protocol-independent model of the network; (2) it models networking boxes using a switch transfer function to support dynamic packets modifications; and (3) it constructs a graph to represent all next-hop dependencies and intra-table dependencies of rules, where all flow paths including both *direct* and *indirect* flow paths in the network can be automatically captured.

Flow Path Space Calculation: For calculating a flow path space, we only abstract fields needed for checking firewall policy violations from the pattern expression of a flow rule to represent the flow path space. In addition, 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 sense, 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 two fields, *destination IP* and *destination port* of the flow rule. Then, we additionally define three kinds of 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 $[P_i^s, P_o^d]$.

3.1.2 Firewall Authorization Space Partition

In many cases, a system administrator may intentionally introduce certain overlaps in firewall rules knowing that only the first rule is important. In reality, this is 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 [26]. 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 introduce an approach, which represents rules with *header space* and performs various set operations on rules, to convert a list of firewall rules into two *disjoint* authorization sub-spaces, *denied authorization space* and *allowed authorization space*. Algorithm 1 shows the pseudocode of partitioning authorization space for a set of firewall rules R. This algorithm works by sequentially examining a header space s_r derived from a rule r and adding 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. Therefore, one 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 $s_{a_i}^F \in S_a^F$, $s_{d_j}^F \in S_d^F$, $1 \leq i \leq n$, and $1 \leq j \leq m$. 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.

3.1.3 Violation Discovery

Once the space of a flow path and the firewall authorization space of the firewall policy are calculated, we identify violations through checking the tracked space (S_t^P) of a flow path, which allows a flow to pass through the network, against the denied authorization space ($S_d^{F'}$) that is a union of all header spaces in the denied authorization space set (S_d^F) of the firewall policy. If these two spaces overlap each other, we call the overlapping space as the 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), which indicates a firewall policy violation. There are two kinds of violations.

- *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, $S_t^P \nsubseteq S_d^{F'}$ and $S_t^P \cap S_d^{F'} \neq \varnothing$.

3.2 Violation Resolution

Since an SDN firewall can directly reject the new flows which violate the firewall policy, it would be straightforward to resolve *flow packet violations*. For resolving *flow policy violations*, an intuitive means is to simply disable the violated flow policies. That is, for a new flow policy, the request for installing a target policy is rejected, if the firewall application detects that the target policy violates the firewall policy. Regarding existing flow policies that violate the firewall policy, they are removed from the network devices directly. However, such a solution have several drawbacks. First, a flow policy may only *partially* violate the firewall policy as we discussed above. 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 a rule in a violated policy may impact other flow policies and even create new violation(s). Obviously, it is necessary to seek a systematic solution to enable a flexible and effective violation resolution. To this end, we introduce a comprehensive violation resolution mechanism, as depicted in Figure 2,

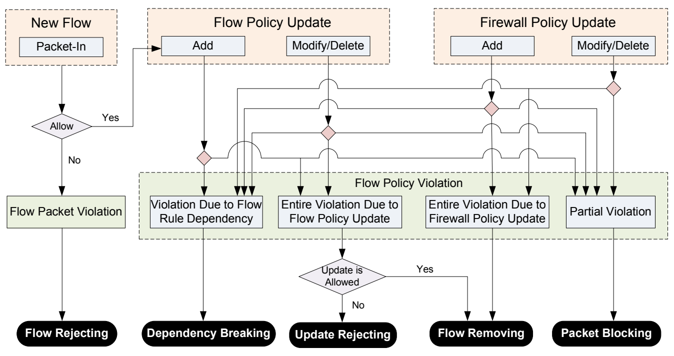


Figure 2: FLOWGUARD violation resolution mechanism.

which demonstrates how FLOWGUARD adopts various violation resolution strategies to resolve different firewall policy violations in terms of new flows and various update operations on both flow and firewall policies in OpenFlow-based networks. We next discuss such a mechanism, focusing on four resolution strategies used for handling flow policy violations.

Dependency Breaking: When a new flow policy is being added into the network switches, this single flow policy may not violate the firewall policy. However, the rules in this new flow policy may overlap with the rules of other existing flow policies. Since rule dependencies could cause unexpected changes in packet headers of flows, they may lead to new firewall policy violations. Note that this kind of violations can be also incurred by other changes of network states, such as modifying flow entries and updating firewall rules. We next introduce two alternative mechanisms for breaking the dependencies among flow rules.

1. *Flow Rerouting*: Using this mechanism, when a new flow comes in the network and the firewall application detects that the flow path generated by the controller for this flow causes a violation due to rule dependencies, the firewall application asks the controller to find another routing path for the flow to avoid those dependencies. During this process, the firewall application maintains a list called *switch evading list* that contains all switches associated with the rule dependencies that cause the violation. The firewall application provides such a list to the controller, and then the controller calculates a new routing path evading those switches in the list. Such a rerouting process may need to be performed recursively until the controller finds a path that does not induce a violation.
2. *Flow Tagging*: Inspired by the network update approach in [20], we explore a *flow tagging* mechanism to break rule dependencies in our violation resolution. In such a 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 takes an additional action on the packets to stamp them with the same tag. As the packets leave the network, in the egress switch, the corresponding rule of the policy strips the tag from the packets.

Using the first mechanism, there are no changes in flow policies and packets. Unfortunately, it may be costly to find a valid flow path. Even, in the worst-case scenario, the controller may not find such a path for a flow that can break all dependencies that cause violations. In contrast, the second mechanism is able to fully resolve the violations. However, it introduces additional processes for changing the flow policies and flow packets. Thus, for some cases, a *hybrid* approach would be desirable, combining both dependency breaking mechanisms to deal with dependency breaking through multiple steps. In the first step, the *flow rerouting* mechanism is applied for breaking dependencies according to a predefined threshold. If the occurrence of rerouting process exceeds the threshold, the violation resolution process automatically jumps to the second step to apply the *flow tagging* mechanism to dependency breaking.

Update Rejecting: There are three possible cases 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 rule induces new *entire* violation(s); and (3) deleting a rule causes new *entire* violation(s), since some rules of other flows have dependency relations with this rule. Applying this strategy, the update operation is rejected directly. Note that, since a change or delete operation on a rule may be mandatory depending on the privileges of the operator, this strategy may be partially applied to cases (2) and (3).

Flow Removing: Two cases can apply this strategy: (1) when updating (adding, changing, or deleting) a rule(s) in the firewall policy, the firewall application examines the current network state applying the updated rule(s) and detects new entire violation(s); and (2) a change or delete operation on a rule is allowed, even though it causes entire violation(s). Using this strategy, all rules associated with a flow path, which entirely violates the firewall policy, are removed from the network switches.

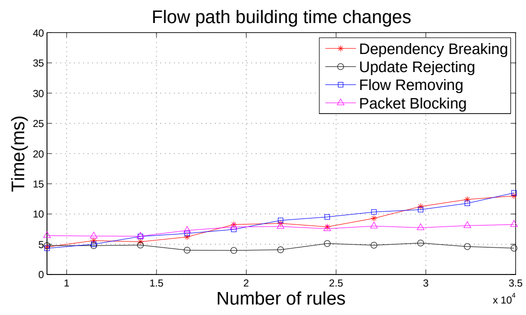
Packet Blocking: For any *partial* violation detected by the firewall application, this strategy can be applied. There may exist two ways to block packets of a flow: (1) if the flow is a new flow, the firewall application only needs to block it in the ingress switch of the flow; and (2) if the flow is an old flow, the firewall application needs to block the packets in both ingress and egress switches. In such a case, blocking packets in the ingress switch can prevent any new packets of the violated flow entering the network, while blocking packets in the egress switch can prevent any *in-flight* packets of the violated flow from going through the network.

In summary, our solution centrally enforces firewall policies to eliminate all flow packet violations and *entire* flow policy violations. However, for *partial* flow policy violation, the *packet blocking* strategy in FLOWGUARD requires that corresponding firewall rules are propagated and enforced in ingress and/or egress switches of violated flow paths. Thus, FLOWGUARD indeed utilizes a *hybrid* architecture to build SDN firewalls and facilitate an effective violation resolution.

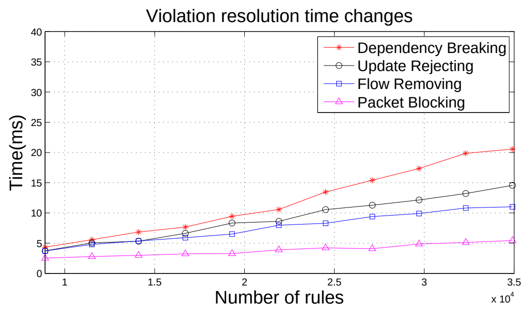
4. IMPLEMENTATION AND EVALUATION

We have implemented a firewall application based on FLOWGUARD on top of Floodlight. Our implementation consists of three components: flow tracking, violation detection and violation resolution. To build flow paths, FLOWGUARD collects network information by monitoring two modules, *Memory Storage Source* and *Static Flow Pusher*, supported by Floodlight controller and employs HSA data structure [3] to compute intra-table dependencies. If the tracked spaces of flow paths overlap with denied authorization space of firewall, FLOWGUARD analyzes the root cause of each violation and leverages a corresponding resolution strategy to resolve the identified violation as illustrated in Figure 2. At the same time, FLOWGUARD maintains updated flow rules and network topology information so that it is able to re-propagate header objects at any associated switches to update flow paths.

We performed experiments based on a *real-world* network topology derived from the Stanford backbone network [11] that includes 14 operational zone Cisco routers, 10 Ethernet switches, and 2 backbone Cisco routers. By using this real-world network, we attempted to evaluate the efficiency and scalability of our firewall application. The entire configuration of the Stanford backbone network was retrieved from [3] and thereby we obtained $1,206$ realistic firewall rules and $8,908$ flow rules in the network.



(a) Flow path building time changes.



(b) Violation resolution time changes.

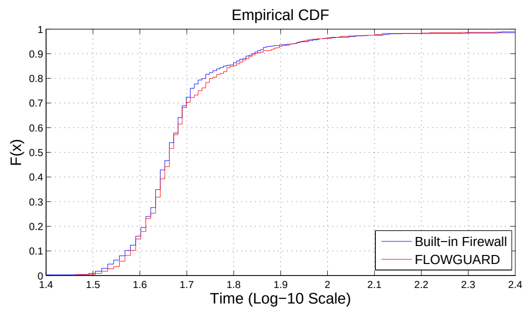
Figure 3: Scalability analysis.

As shown in Table 1, the detection and resolution time for *flow rejecting* strategy were only 0.03 milliseconds (ms). Considering *dependency breaking* strategy, the elapsed time for building flow paths and detecting violations were 4.54 ms and 0.04 ms, respectively. If *flow tagging* mechanism is adopted, FLOWGUARD took 4.34 ms to resolve violations, whereas it only spent, if successful, 1.88 ms resolving violations using *flow rerouting* mechanism. *Update rejecting* and *flow removing* strategies imposed almost similar overheads to the network. Regarding *packet blocking* mechanism, FLOWGUARD took 6.42 ms for building flow paths and 2.53 ms for violation resolution. Since building flow paths for identifying partial violations demands more steps to deal with ingress and egress switches, FLOWGUARD took a little more time for flow tracking in such a case.

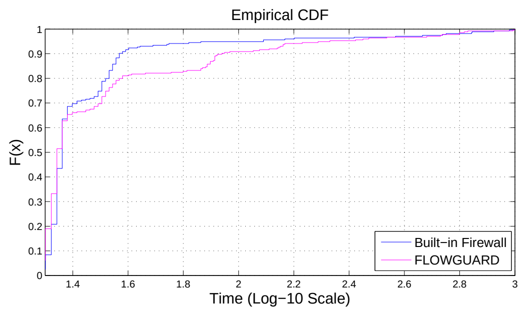
To evaluate the scalability of FLOWGUARD, we increased the number of flow rules based on the Stanford network topology. We only checked the processes for flow tracking and violation resolution, since the violation detection time for all strategies were very close as shown in Table 1. By inserting $100 \sim 1,000$ additional flow rules in each switch, the Stanford network has 8.9 $k \sim 35$ k rules in total since it contains 26 switches. As shown in Figure 3(a), the flow path building time was increased linearly in accordance with the growing number of flow rules except *update rejecting* strategy since it does not cause any changes of flow paths.

Table 1: Measurements of flow tracking, violation detection, and violation resolution time (ms) for different resolution strategies.

	Flow Rejecting	Dependency Breaking		Update Rejecting	Flow Removing	Packet Blocking
		Tagging	Rerouting			
Tracking	-	4.54		4.78	4.32	6.42
Detection	0.03	0.04		0.05	0.07	0.06
Resolution	0.03	4.34	1.88	3.73	3.71	2.53



(a) Firewall rule update time in microsecond.



(b) Per packet inspection time in microsecond.

Figure 4: Performance comparison.

As depicted in Figures 3(b), FLOWGUARD spent less than 25 ms to resolve each violation in the network with a large number of flow rules.

We also compared the performance of FLOWGUARD with the performance of the Floodlight built-in firewall (FW). As shown in Figure 4(a), FLOWGUARD has almost the same update time as FW does under the same network conditions. Most rules could be updated in less than 63 μs in the Stanford topology. As depicted in Figure 4(b), the inspection time of 90% packets for FLOWGUARD was less than 79 μs, while FW spent less than 40 μs inspecting 90% packets. In summary, even though FLOWGUARD took longer to inspect packets than FW, the processing performance was still noticeable.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a comprehensive framework, FLOWGUARD, to facilitate accurate detection as well as flexible resolution of firewall policy violations in dynamic OpenFlow networks. In addition, we have implemented a prototype of FLOWGUARD in Floodlight. Our experimental results show that FLOWGUARD has the manageable performance overhead to enable *real-time* monitoring of SDNs.

As our on-going work, we are currently developing and integrating stateful packet inspection and analysis modules in the FLOWGUARD framework to support the stateful firewall for SDNs. We will also explore various toolkits for supporting visualization, optimization, migration, and integration of SDN firewalls proposed in the FLOWGUARD framework. Furthermore, we plan to integrate our conflict detection and resolution solution into popular SDN controllers to build robust security enforcement kernels for SDN controllers.

Acknowledgments

The work of W. Han, G.-J. Ahn and Z. Zhao was partially supported by the grant from Department of Energy (DE-SC0004308).

6. REFERENCES

[1] Floodlight: Open SDN Controller. `http://www.projectfloodlight.org`.
[2] Frenetic: A Family of Network Programming Languages. `http://frenetic-lang.org/`.
[3] Header Space Library. `https://bitbucket.org/peymank/hassel-public`.
[4] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM'04*.
[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 2014.
[6] B. Braga, M. Mota, P. Passito, et al. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *LCN'10*.
[7] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *NSDI'14*.
[8] H. Hu, G.-J. Ahn, and K. Kulkarni. FAME: a firewall anomaly management environment. In *SafeConfig'10*.
[9] H. Hu, G.-J. Ahn, and K. Kulkarni. Detecting and resolving firewall policy anomalies. *IEEE Transactions on Dependable and Secure Computing*, 9(3):318–331, 2012.
[10] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *CCS'00*.
[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI'13*.
[12] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI'12*.
[13] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *NSDI'13*.
[14] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *HotSDN'13*.
[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM'11*.
[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.
[17] S. A. Mehdi, J. Khalid, and S. A. Khayam. Revisiting traffic anomaly detection using software defined networking. In *RAID'11*.
[18] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *NSDI'13*.
[19] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *HotSDN'12*.
[20] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM'12*.
[21] E. E. Schultz. A framework for understanding and predicting insider attacks. *Computers & Security*, 21(6):526–531, 2002.
[22] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: scalable and vigilant switch flow management in software-defined networks. In *CCS'13*.
[23] S. Shirali-Shahreza and Y. Ganjali. Flexam: Flexible sampling extension for monitoring and security applications in openflow. In *HotSDN'13*.
[24] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: static checking for stateful networks. In *HotMiddlebox'13*.
[25] J. Wang, Y. Wang, H. Hu, Q. Sun, H. Shi, and L. Zeng. Towards a security-enhanced firewall application for openflow networks. In *Cyberspace Safety and Security*, 2013.
[26] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, P. Mohapatra, and C. Davis. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy*.