Security Policy Checking in Distributed SDN based Clouds

Sandeep Pisharody, Ankur Chowdhary and Dijiang Huang
School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, AZ 85281
{spishar1, achaud16, dhuang8}@asu.edu

***Abstract*—Separation of network control from devices in Software Defined Network (SDN) allows for centralized implementation and management of security policies in a cloud computing environment. The ease of programmability also makes SDN a great platform implementation of various initiatives that involve application deployment, dynamic topology changes, and decentralized network management in a multi-tenant data center environment. Dynamic change of network topology, or host reconfiguration in such networks might require corresponding changes to the flow rules in the SDN based cloud environment. Verifying adherence of these new flow policies in the environment to the organizational security policies and ensuring a conflict free environment is especially challenging. In this paper, we extend the work on rule conflicts from a traditional environment to an SDN environment, introducing a new classification to describe conflicts stemming from cross-layer conflicts. Our framework ensures that in any SDN based cloud, flow rules do not have conflicts at any layer; thereby ensuring that changes to the environment do not lead to unintended consequences. We demonstrate the correctness, feasibility and scalability of our framework through a proof-of-concept prototype.**

I. Introduction

Software Defined Networking (SDN) is a transformative approach to network design and implementation based on the premise that separating control of network functions from the network devices themselves (switches, routers, firewalls, etc.) grants network administrators granular control over traffic flow. Using the OpenFlow protocol, SDN switches can leverage the flexibility afforded by the ability to access header information from several layers of the Open Systems Interconnection (OSI) stack. The programmability of SDN can be utilized to respond rapidly to changing user and security requirements by enabling dynamic network reconfiguration. However, this flexibility and programmability also brings to light complex issues with regards to implementing a holistic security infrastructure for an SDN based cloud computing environment. Amongst these are issues caused by flow rule chaining, cross-layer policy conflicts, partial matches, and challenges due to `set-field` actions; all of which are unique to SDN environments. Further, abstracting the data plane from the control plane means that multiple tenants of the cloud could share the same physical network. The users on these cloud environments would control and secure their logical infrastructure by implementing flow rules without any knowledge of the system as a whole. While these flow rules are implementing perceivably private security policies, their presence on a shared control plane means it could lead to potential flow rule conflicts in the overall environment.

Just as firewall conflicts in a traditional network limits the effectiveness of a security infrastructure [1], conflicts between flow rules on the controller limits the effectiveness and impact of a security infrastructure in an SDN based cloud. Unlike a traditional network where new rules can get added only by an administrator, in an SDN based cloud environment, any module running on the controller could introduce new flow rules without having an understanding of existing flow rules or the desired security policy, which could also result in conflicts. To complicate matters further, a dynamically changing network topology adds its own wrinkles.

Substantial research has attempted to address the problems brought forth above, significant amongst which are FortNOX [2] and Flowguard [3] (Section II discusses these further). While these solutions deal effectively with direct policy conflicts and flow violations, they do not tackle a problem that is heightened in an SDN environment - cross-layer policy conflicts. In several multi-tenant clouds implemented using SDN, tenants use flat layer-2 topologies due to latency concerns and the ability to conduct inline promiscuous monitoring using layer-2 devices [4]. A natural extension would be to implement layer-2 flow rule policies. The data center itself might operate in a more traditional fashion with flow rules based on layer-3 addresses. Further, in a traditional environment, a firewall can satisfactorily enforce security policies by having an end-to-end view based on layer-3 addresses. But in an SDN based cloud, policies could be based on layer-2 tunnels over multi-hop networks. If different policy enforcement points enforce policies based on different layers, inconsistent actions could result. Cross-layer conflicts become exacerbated in an SDN based cloud where each SDN switch, both physical and virtual, can be considered to be a distributed firewall instance, with a different local view of the environment and policies.

In our work, we present a formalism for reasoning about conflicts in OpenFlow rules, motivated by the use case of a dynamically changing cloud environment. We begin by classifying all potential conflicts in an SDN environment and introduce a new class of conflicts, *imbrication*, that accounts for indirect cross-layer policy conflicts. Using this formalism, we develop a methodology to conduct temporal mapping between the address ranges of the policies thereby accurately detecting cross-layer conflicts. We develop a controller-based algorithm for detecting conflicts in a flow tables. We further

978-1-5090-3065-1/16/$31.00 ©2016 IEEE

address automatic and assisted conflict resolution mechanisms. This work is implemented in a framework that effectively scrubs the flow table, highlighting and resolving *all* potential conflicts. Moreover, we account for the possibility that in a multi-tenant cloud environment, not all parties would play fair, with some attempting to prioritize *their* security policy - a problem our framework effectively addresses. To summarize, in this work we classify, detect and resolve flow rule conflicts in an SDN based distributed cloud environment including cross-layer policy conflicts. Correctness was verified on a test network with 100 rules in its flow table and scalability was tested on the Stanford backbone rule set [5], extrapolated upto $100,000$ rules. Performance impact of running our module on the controller was also evaluated, and found to be within an acceptable 5% overhead.

This paper is organized as follows. We discuss related work in Section II. In Section III we lay out some background information about flow rule conflicts, produce a real-world motivating scenario for our framework along with formalizing and classifying flow rule conflicts. In Section IV we discuss our system design and implementation details. Evaluation information is discussed in Section V. Finally, we conclude and comment on our future research tasks in Section VI.

II. Related Work

There have been several attempts at a security solution in SDN. The most basic SDN firewall is introduced as part of Floodlight [6], where each new packet in a flow is sent to the controller, where it is matched against a set of rules. The resulting action set is then sent to the OpenFlow switch for implementation for the current and future matching flows. In case of dynamic security policy updates to the controller, the OpenFlow switches could be left implementing a dated action set. Javid et al. [7] built a layer-2 firewall for an SDN environment using a tree topology for a small network using a POX controller and restricted traffic flow as desired. Suh et al. [8] illustrated a proof-of-concept version of a traditional firewall over an SDN controller. But neither of these works addressed the problem of conflicting flow rules.

Pyretic [9] deals effectively with direct policy conflicts, by placing them in a prioritized rule set much like the OpenFlow flow table. However, indirect security violations or inconsistencies in a distributed SDN environment cannot be handled by Pyretic without a flow tracking mechanism such as the one discussed by Fayazbakhsh et al. [10]. FRESCO [11] allows for the implementation of security services in an OpenFlow environment by providing reusable modules accessible through a Python API. To address conflicts that might arise in an OpenFlow environment, FRESCO introduces a Security Enforcement Kernel (SEK) that prioritizes rules to assist in conflict resolution.

FortNOX [2] is an extension to the NOX controller that implements role-based and signature based enforcement to ensure modules do not circumvent the existing security policy. In FortNOX, reusable modules are used to protect the flow installation mechanism against adversaries, but conflict analysis is conducted only between a new flow rule and existing rules, without considering dependencies within flow tables. Thus, implementing FortNOX in a distributed environment would be challenging. Decision making in FortNOX seems to follow a least permissive strategy instead of making a decision keeping the holistic nature of the environment in mind. Moreover, it uses only IP and port information for conflict detection, which we believe is incomplete since SDN flow rules could use purely layer-2 addresses for decision making. In addition FortNOX would not be able to handle partial flow rule conflicts.

Flowguard [3] is a security tool specifically designed to resolve security policy violations in an OpenFlow network. Flowguard examines incoming policy updates and determines flow violations in addition to performing stateful monitoring. It uses several strategies to refine anomalous policies, most of which include rejecting a violating flow.

VeriFlow [12] is a proposed layer between the controller and switches which conducts real time verification of rules being inserted. It verifies that flow rules being implemented have no errors due to faulty switch firmware, control plane communication, reachability issues, configuration updates on the network, routing loops, etc. However, none of the solutions discussed tackle what we believe is a problem that is heightened in an SDN environment - cross-layer policy conflicts in distributed environments. To that end, we propose a tool that will consider cross-layer dependencies while ensuring conflict-free policies in a SDN based distributed cloud environment.

III. Background & Model

A. Motivating Scenario

Implementing a management system that only specifies security policies without tackling topological interaction amongst constituent members has always been a recipe for conflicts [13]. With the SDN controller having visibility into the entire system topology along with the policies being implemented, several of the conflict causing scenarios in traditional networks were handled. However, there are several instances where conflicts can creep into the flow table such as *a*) service chain processing where multiple flow tables that handle the same flow might have conflicting actions; *b*) VPN implementations that modify header content could result in flow rules being inadvertently being applied to a certain flow; *c*) flow rule injection by different modules (using the northbound API provided by the controller) could have conflicting actions for the same flow; *d*) matching on different OSI layer addresses resulting in different actions; and *e*) administrator error. This list, while incomplete, goes to show how prevalent policy conflicts in SDN environments could be. We illustrate a L2VPN based scenario to serve as a real life motivating example.

In a multi-tenant hosted data center cloud, the provider could have layer-3 rules in place to prevent certain tenants from sending traffic to one another for monetization, compliance or regulatory reasons. Hosts in two different tenant environments, Tenant A and Tenant B, can establish a layer-2 tunnel (either as a host-to-host tunnel or a site-to-site tunnel) between themselves to do single hop computation or to encrypt communication between them as shown in Figure 1. To accommodate this VPN, flow rules will be inserted on the controller that permit traffic between the layer-2 addresses of servers on Tenant A and Tenant B; which clearly conflict with the earlier

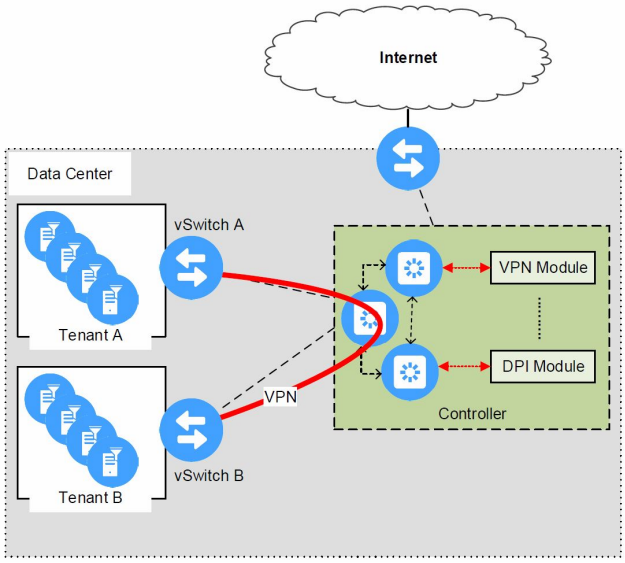

Fig. 1. Motivating scenario for cross-layer conflicts.

policy of not allowing traffic between the layer-3 addresses of Tenant A and Tenant B. Further, if a policy is inserted into the controller to implement Deep Packet Inspection (DPI) on all packets exiting Tenant A, all traffic from Tenant A to Tenant B will be dropped, since they are encrypted and fail the DPI standards. Clearly there is an inherent conflict between flow rules inserted by different modules that are running on the SDN controller, leading to a shoddy user experience.

B. Flow Rule Management Challenges

Security implementations using SDN leverage the ability to make dynamic changes to the network and system configurations to have a lean, agile and secure environment. Since this usually results in environments that are constantly in flux, any kind of a security infrastructure has to detect these changes in real time, and update flow rules on the (distributed) controllers to keep it in line with the security policy of the organization.

Since flow rules have the ability to match more than just layer-3 and layer-4 headers as in a traditional network, they are inherently more complex by virtue of having additional variables to consider for a match. The actions that can be applied on a match include forwarding to specific ports on the switch, flooding the packet, changing its QoS levels, dropping the packet, encapsulating, encrypting, rate limiting or even customizable actions using various `set-field` actions. Since cross-layer interaction is bolstered in SDN by virtue of having flow rules that permit `set-field` actions, several packet headers can be dynamically changed. And lastly, since wildcard rules are allowed, a partial conflict of a flow policy could occur, thereby adding complexity to the resolution of conflicting flow rules.

As opposed to a traditional network, flow rules in SDN, could have the same priorities as well as matches on multiple header fields, thereby resulting in indirect dependencies. For example, consider traffic originating from host A destined to host C in Figure 5. This flow would clearly match rule 8 in Table I based off a layer-2 address match; and rule 1 based off a layer-3 address match. A flawed approach to tackle this problem would be to expand the header space and determine rule conflicts as in a traditional environment since there exists an indirect dependency between the layer-2 and layer-3 addresses. Moreover, flow rules could exist that do not include all the header fields making an apples-to-apples comparison impossible. We take a different approach to detecting and solving indirect conflicts (discussed in Section III-C). Since these conflicts arise as a result of addressing across different OSI layers, we categorize them accordingly and tackle it as such.

C. Conflict Classification

We first formally define the set operations on address spaces. Let ξ be a 2-tuple address space (ξ_s, ξ_d), with subscript s denoting the source address set and d denoting the destination address set. Then the following definitions apply.

Definition 1. *An address space $\xi_i \subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{si} \subseteq \xi_{sj} \wedge \xi_{di} \subseteq \xi_{dj}$.*

Definition 2. *An address space $\xi_i \not\subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{si} \not\subseteq \xi_{sj} \vee \xi_{di} \not\subseteq \xi_{dj}$.*

Definition 3. *An address space $\xi_i \subset \xi_j$ if and only if they refer to the same OSI layer, and $(\xi_{si} \subset \xi_{sj} \wedge \xi_{di} \subseteq \xi_{dj}) \vee (\xi_{si} \subseteq \xi_{sj} \wedge \xi_{di} \subset \xi_{dj})$.*

Definition 4. *Address space intersection $\xi_i \cap \xi_j$ produces a tuple $(\xi_{si} \cap \xi_{sj}, \xi_{di} \cap \xi_{dj})$ if and only if ξ_i and ξ_j refer to the same OSI layer. When ξ_i and ξ_j are not addresses in the same OSI layer, the $\cap$ operation is invalid.*

Rules present in a flow table consist of *a*) rule priority; *b*) match fields consisting of header information and ingress port that is used to match flows; *c*) packet counters; *d*) the action set; and *e*) timeouts. Of these fields, we focus on just priority, match fields and actions to handle flow rule conflicts. Table I shows sample flow table rules with the relevant fields present. Ingress port and VLAN fields have been ignored in this example.

Since flow table rules in an SDN environment closely match firewall rules, we extend the work done on firewall rule conflicts. Although there have been several attempts to classify firewall rule conflicts [13], [14], [15], [16], the seminal work by Al-Shaer and Hamed [17] is often used to classify firewall rule conflicts in a single firewall environment. We build on that work to formally classify flow rule conflicts.

In a flow table F containing rule set $\{r_1, r_2, ..., r_n\}$; we can represent $r_i = (p_i, \lambda_i, \epsilon_i, \zeta_i, \eta_i, \varphi_i, \rho_i, a_i)$, where *a*) p is the priority of the rule and is defined in the range $[1, 65535]$; *b*) λ is the ingress port; *c*) ϵ is a 2-tuple (ϵ_s, ϵ_d) OSI layer-2 address space, with subscript s denoting source and d denoting destination address; *d*) ζ is a 2-tuple (ζ_s, ζ_d) OSI layer-3 address space, with subscript s denoting source and d denoting destination address; *e*) η is a 2-tuple (η_s, η_d) OSI layer-4 address space, with subscript s denoting source and d denoting destination address; *f*) ρ is the layer-4 protocol; and *g*) action

Rule #	Priority	Source MAC	Dest MAC	Source IP	Dest IP	Protocol	Source Port	Dest Port	Action
1	51	*	*	10.5.50.0/24	10.211.1.63	tcp	*	*	permit
2	50	*	*	10.5.50.5	10.211.1.63	tcp	*	80	permit
3	52	*	*	10.5.50.5	10.211.1.0/24	tcp	*	*	permit
4	53	*	*	10.5.50.0/24	10.211.1.63	tcp	*	*	deny
5	54	*	*	10.5.50.5	10.211.1.63	tcp	*	*	deny
6	51	*	*	10.5.50.0/16	10.211.1.63	tcp	*	*	deny
7	55	*	*	10.5.50.5	10.211.1.0/24	tcp	*	80-90	deny
8	57	11:11:11:11:11:ab	11:11:aa:aa:11:11	*	*	*	*	*	permit
9	58	*	*	*	*	tcp	*	80	deny

TABLE I. FLOW TABLE EXAMPLE.

a is the action set for the rule.

Knowing that OpenFlow specifications clarify that if a packet matches two flow rules, only the flow rule with the highest priority is invoked, the different flow rule conflicts are categorized as below:

- **Redundancy:** A rule r_i is redundant to rule r_j iff *a*) address space $\epsilon_i \subseteq \epsilon_j \wedge \zeta_i \subseteq \zeta_j \wedge \eta_i \subseteq \eta_j$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i = a_j$. For example, rule 2 in Table I has a packet space that is a subset to the packet space of rule 1, with matching protocol and actions. Hence, rule 2 is redundant to rule 1. Redundancy does not pose a serious issue, but instead is more of an optimization and efficiency problem.
- **Shadowing:** A rule r_i is shadowed by rule r_j iff *a*) priority $p_i < p_j$; *b*) address space $\epsilon_i \subseteq \epsilon_j \wedge \zeta_i \subseteq \zeta_j \wedge \eta_i \subseteq \eta_j$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$. In such a situation, rule r_i is never invoked since incoming packets always get processed using rule r_j, given its higher priority. Shadowing is a potentially serious issue since it shows a conflicted security policy implementation [17]. For example, rule 4 in Table I has the same packet space as rule 1, with the same protocol, but conflicting actions. Hence, rule 1 is shadowed by rule 4.
- **Generalization:** A rule r_i is a generalization of rule r_j iff *a*) priority $p_i < p_j$; *b*) address space $(\epsilon_i \supset \epsilon_j \wedge \zeta_i \supseteq \zeta_j \wedge \eta_i \supseteq \eta_j) \vee (\epsilon_i \supseteq \epsilon_j \wedge \zeta_i \supset \zeta_j \wedge \eta_i \supseteq \eta_j) \vee (\epsilon_i \supseteq \epsilon_j \wedge \zeta_i \supseteq \zeta_j \wedge \eta_i \supset \eta_j)$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$. In this case, the entire packet space of rule r_j is matched by rule r_i [17]. As shown in Table I, rule 1 is a generalization of rule 5, since the packet space of rule 5 is a subset of the packet space of rule 1, and the actions are different. Note that if the priorities of the rules are swapped, it will result in a shadowing conflict.
- **Correlation:** Classically, a rule r_i is correlated to rule r_j iff *a*) priority $p_i < p_j$; *b*) address space $\epsilon_i \not\subseteq \epsilon_j \wedge \zeta_i \not\subseteq \zeta_j \wedge \eta_i \not\subseteq \eta_j \wedge \epsilon_i \not\supseteq \epsilon_j \wedge \zeta_i \not\supseteq \zeta_j \wedge \eta_i \not\supseteq \eta_j \wedge (\epsilon_i \cap \epsilon_j \neq \emptyset \vee \zeta_i \cap \zeta_j \neq \emptyset \vee \eta_i \cap \eta_j \neq \emptyset)$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$ [17]. As shown in Table I, rule 3 is correlated to rule 4. Since multiple SDN flow rules can have the same priority, we make the following addition to the correlation conflict: A rule r_i is correlated to rule r_j iff *a*) priority $p_i = p_j$; *b*) address space $\epsilon_i \cap \epsilon_j \neq \emptyset \vee \zeta_i \cap \zeta_j \neq \emptyset \vee \eta_i \cap \eta_j \neq \emptyset$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$. Thus the correlation conflict now encompasses all policies that have the different actions, overlapping address spaces and the same priority. For example, in Table I, rule 6 is correlated to rule 1.
- **Overlap:** A rule r_i overlaps rule r_j iff *a*) priority $p_i \leq p_j$; *b*) address space $\epsilon_i \not\subseteq \epsilon_j \wedge \zeta_i \not\subseteq \zeta_j \wedge \eta_i \not\subseteq \eta_j \wedge \epsilon_i \not\supseteq \epsilon_j \wedge \zeta_i \not\supseteq \zeta_j \wedge \eta_i \not\supseteq \eta_j \wedge (\epsilon_i \cap \epsilon_j \neq \emptyset \vee \zeta_i \cap \zeta_j \neq \emptyset \vee \eta_i \cap \eta_j \neq \emptyset)$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i = a_j$. An overlap rule is essentially a correlation; but with the same action set. This overlap can be seen between rule 6 and rule 7 in Table I.
- **Imbrication:** The criteria discussed above does not cover all potential conflicts. Consider the case of flow rules, *a*) only layer-3 header is used as a condition (rule 1-7); *b*) only layer-2 header is used as a condition for decision (rule 8); and *c*) only layer-4 header is used as a condition (rule 9). Even though using our definition there is no overlap in packet space, and hence there should be no conflict, a packet could match more than one of these rules. We classify such policy conflicts as imbricates, and address them by introducing the concept of *reconciliation* (described in Section IV) which maps all headers to the same layer. Using the topology shown in Figure 5 and the flow rules in Table I, we can see flow rule 4, which denies traffic from host A to host D and flow rule 8, which permits traffic from host A to any other host are clearly imbricates.

IV. SYSTEM DETAILS

A. System Modules

Our framework as shown in Figure 3 is an intuitive model to help resolve conflicts in flow rules in a distributed SDN environment. It consists of several inter-related modules that together achieve a conflict free flow table. It runs as an application on the controller that listens for new/modified flow rules from different applications running on the controllers. The processing is compartmentalized to sanitization, conflict detection and conflict resolution. The modules that accomplish these tasks are:

- *Flow rule monitor:* The flow rule monitor intercepts any

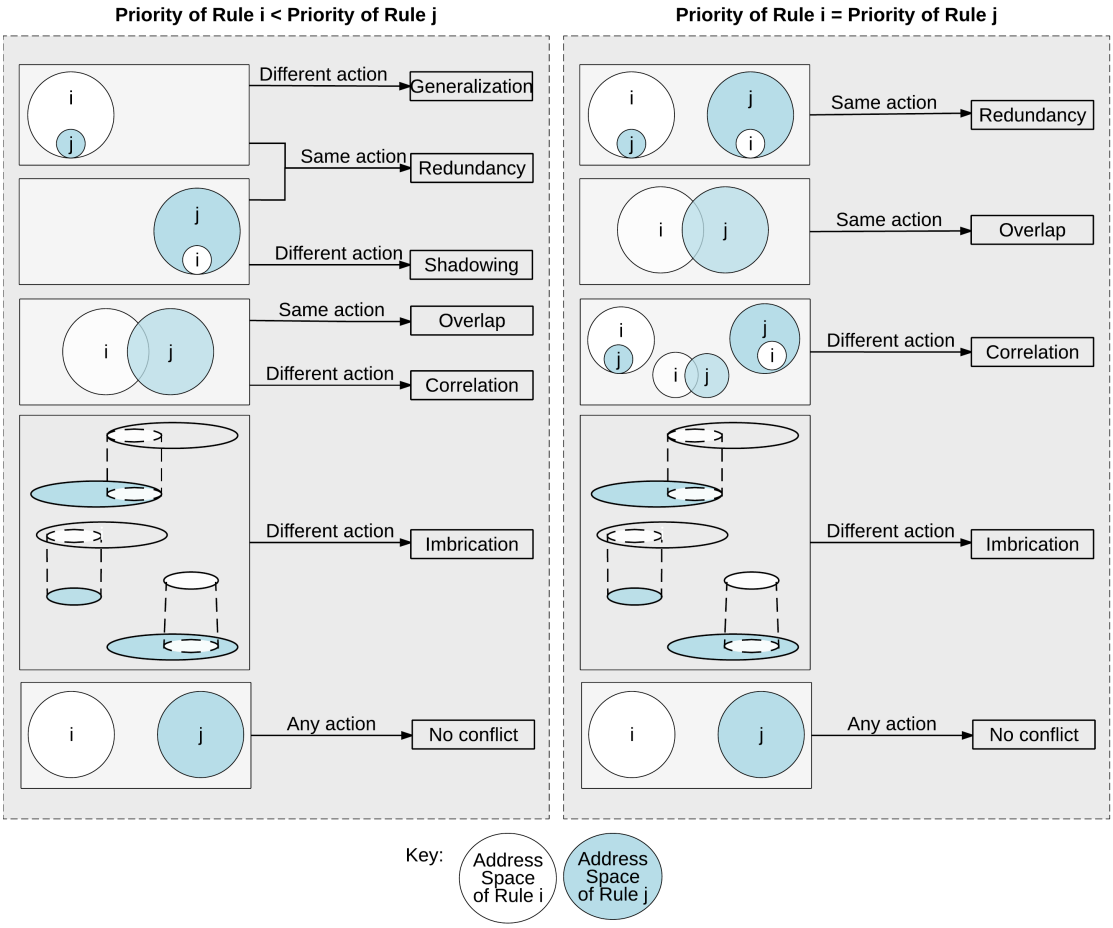

Fig. 2. Venn diagram showing address space overlap and rule conflicts.

new or updated flow rule that is to be added to the flow table. These rules, which we call *candidate* flow rules, can be generated by any module running on the controller or by the administrator. A candidate flow rule is not completely processed and vetted, and hence is not eligible to be sent to any of the devices. In a distributed controller scenario, candidate flow rules into every controller is obtained, in an effort to have complete knowledge of all possible flow rules that are present in the environment. The priority of the rules from the controller are modified to a global priority, based on the decentralization strategy that has been employed. Thus, the priority assigned by the flow rule monitor may differ from the priority of the flow rule present in the flow table. In a redundant controller setup, only candidate rules from the master controller are obtained, and in a truly distributed environment, candidate flow rules are aggregated from all the controllers before processing.

- *Atomization engine:* Since OpenFlow permits chained flow rules by having an action for a match redirect to a different flow table, in order to correctly identify conflicts between flow rules, we *atomize* the flow rules by processing the chains and ensuring that only the atomic actions of permit and deny remain. The atomization process itself follows along the lines of `ipchain` processing in Unix. There are two important considerations we make here.
  - While the actions for a flow rule can include any drop, forward, flood, set QoS parameters, change several header fields, or redirect to a different flow table; we process the actions and generically classify them into two categories; permit and deny. For example, implementing an IP mapping rule in OpenFlow would change the IP address headers and forward onto a different flow table that forwards the traffic. We process such a chain to include the address translation information and set the final atomic action to be permit.
  - For rules which have multiple actions, we duplicate the rules to generate rules with identical priority and match conditions with a single action.
- *Conflict detection module:* This module identifies conflicts based on the categories described in Section III-C. Special attention is given to rules which have only layer-2 match conditions by mapping them to their layer-3 addresses using a process we call *reconciliation*. The mapping information is obtained using a temporal 1-to-1 mapping by doing a table lookup. In cases where a

mapping is found, we tag the rule indicating a reconciled address to identify flow rules which fall into the imbrication conflict. Rules that have only layer-4 match conditions are also tagged as such.

- *Conflict resolution module:* Once the flow rule conflicts have been detected, the conflict resolution module is invoked. Most conflicts are resolved automatically. However, in case of interpretative conflicts (Generalization, Correlation and Imbrication) which cannot be resolved automatically, resolution strategies discussed in Section IV-C are used.

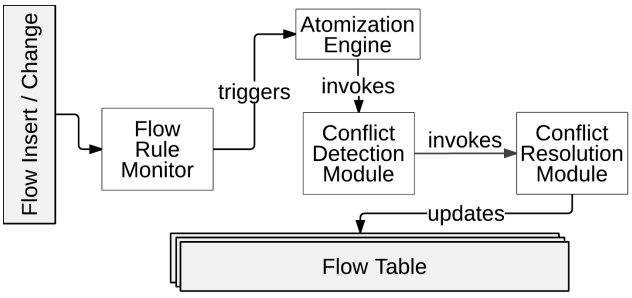

Fig. 3. System overview representing different constituent modules.

B. Conflict Detection Mechanism

The default OpenFlow rule specifications do not provide us all the information we need to detect and resolve flow rule conflicts. Thus we add a data structure to accompany the existing OpenFlow rule using four additional fields over 32 bits of information as shown in Figure 4. These fields are: *a*) One bit identifying if the rule in question has been tagged as a reconciled rule (required for imbricate detection); *b*) seven bits identifying the SDN controller to which the rule is going to be inserted; *c*) sixteen bits for a global priority of the flow rule (to be used for flow rule conflict resolution); and *d*) eight bit Conflict Resolution Criteria (CRC) metric (discussed in Section IV-C). Armed with these additional bits of information, we can now detect flow rule conflicts using the methodology shown in Algorithm 1.

During environment ramp up, the flow table on the

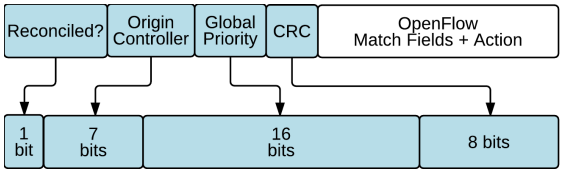

Fig. 4. Data structure format.

controller is parsed, and *atomized* by the atomization engine. As a result of this process, every flow rule that has an action which leads to execution of a different flow table is modified into one or more rules that have terminal actions of permit, deny, QoS and packet count. Since QoS and packet counting can be processed along with the permit and deny actions, these rules are removed from further processing.

The conflict detection module *reconciles* all the layer-2 only rules by doing an address mapping to layer-3, and marks them as a reconciled rule. Since we formally describe any overlaps involving reconciled rules as imbricates, we process those rules separately from non-reconciled rules and classify them as such. The conflict detection module then employs a Patricia trie [18] lookup to detect address space overlap.

The Patricia trie is an efficient search structure for finding matching IP strings [19] with a good balance between running time (lookup and update) and memory space requirement, and has been used previously with great success [20], [21]. Since we know that the layer-3 addresses are fixed length, we can follow along a path from the root to a matching node to obtain flow entries that match the address space of the flow being processed. In cases of wildcard matches, all child nodes of the matching node will represent flow entries conflicting with the input flow. We do an octet wise Patricia trie lookup [19] to look for IP address range overlap between the new rules being inserted and existing rules in the flow table in a fast and efficient manner. All detected conflicts are classified as shown in Figure 2. Next, the conflict resolution module is invoked. Details on the resolution strategies are provided in Section IV-C.

C. Conflict Resolution Strategies

Once the conflicts between different flow rules have been detected, the conflict resolution module attempts to resolve these. The different conflict types can be broadly categorized into *Intelligible* and *Interpretative* conflicts. The resolution strategies for each of these two categories are markedly different, as seen in Algorithm 2.

1) Intelligible Conflicts: Since flow rules that conflict with each other in the Redundancy, Shadowing and Overlap classifications all have the same action they can be resolved without the loss of any information. In other words, any packet that is permitted by the controller prior to resolving the conflict will continue to be permitted after conflict resolution.

Intelligible conflicts are resolved fairly easily by eliminating the rules that are not applied, or by combining and optimizing the address spaces in the rules so as to avoid the conflict. It could be argued that creative design of rules by administrators result in flow rules that deliberately conflict so as to optimize the number of rules in the flow table, especially when it comes to traffic shaping policies. However, such optimization strategies stem out of legacy network management techniques, and do not hold true in dynamic, large-scale cloud environments where the flow table enforcing the policies in the environment could have millions of rules.

2) Interpretative Conflicts: Conflicts that fall into the Generalization, Correlation and Imbrication classifications can not be intuitively resolved without loss of some information, and are interpretative in nature. As opposed to intelligible conflicts, it is not guaranteed that any packet permitted by the controller prior to resolving the conflict will be permitted after conflict resolution. Since interpretative conflict resolution is lossy in nature, the resolution strategies are *not* a one size fits all and need to be adapted according to the cloud environment in question. We discuss a few resolution strategies that could

Algorithm 1 Conflict Detection Module

```
1:  procedure FLOWCONFLICTS(FlowTable f)
2:  Input: Rule r, FlowTable f
3:  Output: Conflict-free FlowTable f'
4:      ATOMIZE(r)
5:      if r is layer-2 rule || layer-4 rule then
6:          RECONCILE(r)
7:      if r doesn't have Reconciled tag then
8:          γ ← SearchPatricia(L3Addr(r))
9:          if Protocol(r)! = Protocol(γ) then
10:             f' ← InsertFlow(f, r) return f'
11:         if Addr(r) ⊆ Addr(γ) then
12:             if Action(r) = Action(γ) then
13:                 ConflictResolve(r, γ, Redundancy)
14:             else
15:                 if Priority(r) == Priority(γ) then
16:                     ConflictResolve(r, γ, Correlation)
17:                 else
18:                     ConflictResolve(r, γ, Generalization)
19:         if Addr(γ) ⊆ Addr(r) then
20:             if Action(r) = Action(γ) then
21:                 ConflictResolve(r, γ, Redundancy)
22:             else
23:                 if Priority(r) == Priority(γ) then
24:                     ConflictResolve(r, γ, Correlation)
25:                 else
26:                     ConflictResolve(r, γ, Shadowing)
27:         if Addr(r) ∩ Addr(γ)! = ∅ then
28:             if Action(r) = Action(γ) then
29:                 ConflictResolve(r, γ, Overlap)
30:             else
31:                 ConflictResolve(r, γ, Correlation)
32:         Insert r to the PatriciaTrie and flow table
33:     else
34:         // Rules with Reconciled tag
35:         for Rule γ in f do
36:             if Protocol(r) == Protocol(γ) then
37:                 if Addr(r) ∩ Addr(γ)! = ∅ then
38:                     ConflictResolve(r, γ, Imbrication)
39:     Insert r to the PatriciaTrie and flow table
```

Algorithm 2 Conflict Resolution Module

```
1:  procedure CONFLICTRESOLVE(Rule r, Rule γ, Conflict-
    Type)
2:
3:      if ConflictType == Shadowing || ConflictType ==
    Redundancy then
4:          // No need to add these rules to the table
5:      if ConflictType == Correlation then
6:          if CRC(γ) > CRC(r) then
7:              Addr(r) ← Addr(r) − Addr(r ∪ γ)
8:              Insert r to the PatriciaTrie and flow table
9:          else
10:             Remove γ from PatriciaTrie and flow table
11:             Addr(γ) ← Addr(γ) − Addr(r ∪ γ)
12:             Insert γ, r to the PatriciaTrie and flow table
13:     if ConflictType == Generalization then
14:         if CRC(γ) > CRC(r) then
15:             Addr(r) ← Addr(r) − Addr(r ∪ γ)
16:             Insert r to the PatriciaTrie and flow table
17:         else
18:             Remove γ from PatriciaTrie and flow table
19:             Insert r to the PatriciaTrie and flow table
20:     if ConflictType == Overlap then
21:         Addr(r) ← Addr(r ∪ γ)
22:         Remove γ from PatriciaTrie and flow table
23:         Insert r to the PatriciaTrie and flow table
24:     if ConflictType == Imbrication then
25:         if CRC(γ) > CRC(r) then
26:             // No need to add these rules to the table
27:         else
28:             Insert r to the PatriciaTrie and flow table
```

be applied. Algorithm 2 makes a generic reference to a Conflict Resolution Criteria (CRC) which is dependent on the resolution criteria in use. The data structure we suggested as a concomitant to the flow rules also refers to this CRC metric.

- Least privilege - In case of any conflict, flow rules that have a deny action are prioritized over a QoS or a forward action. If conflicts exist between a higher and lower bandwidth QoS policy, the *lower* QoS policy is enforced. The least privilege strategy is traditionally one of the most popular strategies in conflict resolution [22].
- Module security precedence - Since flow rules in an SDN based cloud environment can be generated by any number of modules that run on the controller, an effective strategy that can be put in place is to have a security precedence for the origin of the flow rule [2]. Thus a flow rule originating from a security module is prioritized over flow rule from an traffic management or optimization module.
- Environment calibrated - This strategy incorporates learning strategies in the environment to make an educated decision on which conflicting flow rule really needs to be prioritized. Over time, if a picture can be formed about the type of data that a certain tenant usually creates/retrieves, or of the applications and vulnerabilities that exist in the tenant environment, or the reliability of the software modules inserting the flow rule; the conflict resolution module may be able to prioritize certain flow rules over others. However, these techniques falter while dealing with a dynamic cloud, especially with Moving Target Defense (MTD) based defensive strategies.
- Administrator assistance - Administrators that are willing to give up automatic conflict resolution have the option to be able to use their infinite wisdom to resolve conflicts. This technique assumes that the administrator knows what is best for the cloud environment. In addition to requiring manual intervention to resolve conflicts, this method is also susceptible to insider poisoning.

V. EVALUATION

The modules described in Section IV were implemented in JAVA, using a command line interface in its current iteration. OpenDaylight (ODL) [23] Lithium was used as the OpenFlow controller and the `L2Switch` project was employed

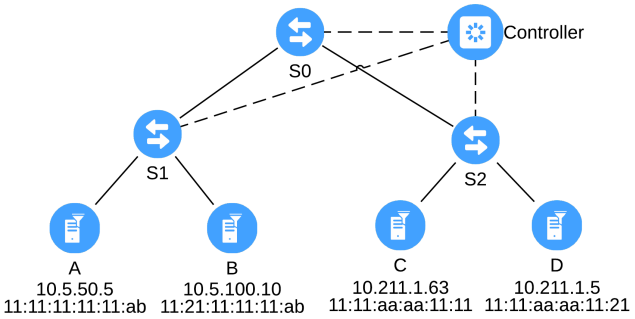

Fig. 5. Test network topology.

to connect to the Open vSwitch (OVS) switches. OVS and ODL Lithium support both OpenFlow 1.0 and OpenFlow 1.3. Our implementation correctly identifies flow rule conflicts and classifies them, including temporal cross-layer conflicts. Both intelligible and interpretative conflicts are automatically resolved using the least privilege resolution strategy.

A simple network with topology shown in Figure 5 was implemented on Mininet to instantiate Open vSwitch switches and virtual hosts using a python script. The flow table rules shown in Table I were used to constrain traffic in the network. Since the flow rules in the table were implemented explicitly with different priorities, traffic flow was as expected. When we processed the flow rules using our framework, it was correctly able to identify several flow rule conflicts, and correct them. Here, we compared our framework with closest related works such as FortNox [2] and FlowGuard [3], and determined that 20% of the conflicts we detected were imbrication conflicts, which would not have been detected by earlier works.

In the next phase of our evaluation, we tested the scalability of system. Here, we performed experiments based on a real-world network topology derived from the Stanford network as obtained by Kazemian et al. [5]. Both the conflict detection and resolution algorithms grow in a linear fashion, except for the Patricia trie lookup and insertion time. The time complexity of a lookup on a Patricia trie depends on the length of the string (constant in our case) and the number of flow rules; for a total runtime of $\mathcal{O}(n)$ [18], where n is the number of entries in the flow table. This result was verified experimentally using a 2.5 GHz Intel Core i7 machine with 16 GB DDR3 memory. With an input file containing about $10,000$ atomic flow rules, the processing time was about 6.45 ms. Rules were further replicated and inserted into the system to observe growth of computation time. Figure 6 shows results from our experiment runs using different input flow table sizes. Ten different test runs were conducted on flow tables of size varying from $10,000$ to $100,000$ rules, and the resulting running times were averaged to get the results in the plot. The results clearly show that our system effectively identifies flow rule conflicts and takes corrective action in spite of the large data sets. The results also clearly show a $\mathcal{O}(n)$ running time. Comparative running times for Flowguard are obtained from [3]. Run times for FortNox are not available and the algorithm complexity is not discussed, but evaluation appears to suggest linear growth; albeit considerably slower (approximately 7 ms per $1,000$ flow rules, as opposed to 0.56 ms per $1,000$ flow rules for our system).

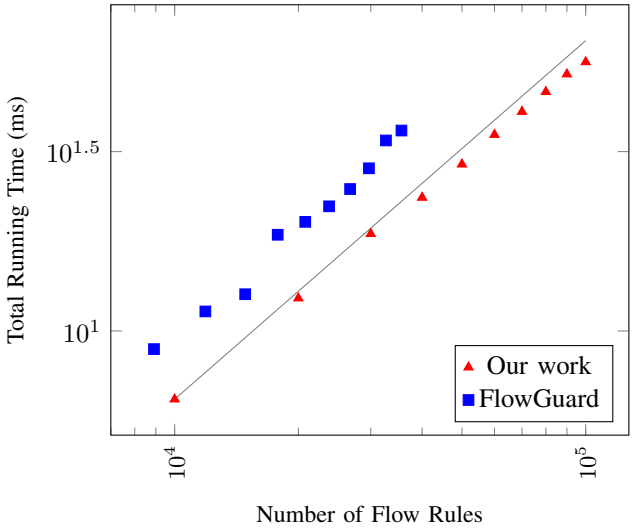

Fig. 6. Running time evaluation.

Once correctness of our work was verified and validated, we analyzed the performance overhead of conducting inline rule conflict analysis. Once again, the topology shown in Figure 5 was used for the experiment. The different link bandwidth were enforced using the `tc` command on Linux. This setup allows us a fine control on the network. A very large file (1 GB) was sent from host A to host D, with a script attempting to add flow rules into the environment. Figure 7 shows the time taken to transfer the file in cases where the rules being inserted were *a*) conflict free; *b*) rules had conflicts that could be automatically resolved; and *c*) conflicts needed administrator assistance for resolution. As expected, when administrator assistance was required, the transfer took longer, due to system resources on the VM being diverted for I/O purposes. Further granular introspection into the data showed that shadowing and redundancy conflicts had the least impact on latency, only because they were the first to be identified in the chained processing. Implementing our system caused about 5% increase in transfer time (average of 100 test runs). We contend that this tradeoff is acceptable in an SDN environment since having a conflict free flow table will not only ensure greater confidence in security, but also more optimal packet forwarding processing times.

VI. Future Work & Conclusion

Traditional approaches to addressing security issues in a dynamic, distributed cloud environment concerned themselves with implementing security on individual components, and not considering security holistically. With the growing size of such environments and its increasingly distributed nature; implementing consistent and conflict-free security policies is progressively challenging. A complete security solution for an SDN based cloud computing environment is incomplete without a set of conflict free flow rules implementing the security policy in the cloud.

In this work, we introduce a framework, that monitors and maintains a conflict free environment. We first present a formalism for the classification of the different types of

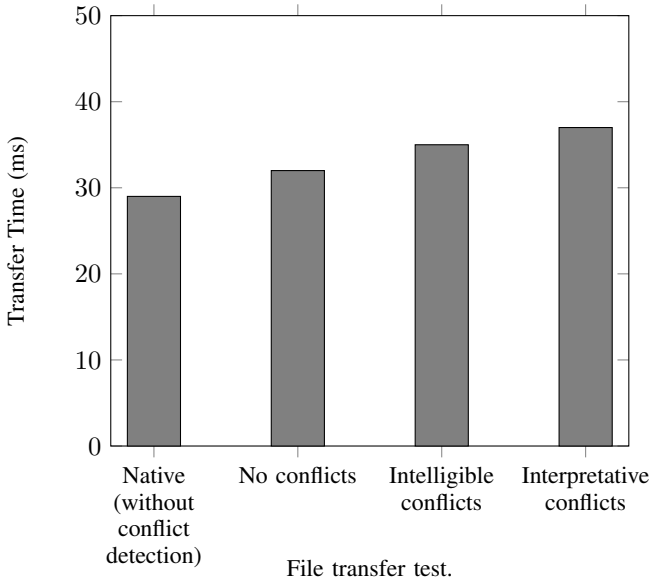

Fig. 7. Network performance impact.

conflicts in SDN, and develop ways to resolve them automatically. When a new flow is added, our system helps maintain conflict-free consistent flow rules among multiple distributed controllers. The run time complexity for the framework is linear, and hence scalable to large data center environments. As part of out next steps, we plan to study using multiple analyzers to share the work load in an effort to parallelize processing. Including role-based and attribute-based policy conflicts is a natural extension of this work. Further, we are looking into flow rule optimization based on rule positioning and examine adaptive prioritization of rules. Incorporating stateful functionality into the current framework is also being studied. Since a one-size fits all solution rarely works, we are also looking into flavors of our framework tailored for host based SDN firewalls and a mobile (lightweight) SDN based cloud. Finally, we plan on expanding the framework to work in an environment with diverse controllers.

ACKNOWLEDGEMENT

This research is supported by NSF Secure and Resilient Networking (SRN) Project (1528099) and NATO Science for Peace & Security Multi-Year Project (MD.SFPP 984425). S. Pisharody is supported by a scholarship from the NSF CyberCorps program (NSF-SFS-1129561).

REFERENCES

[1] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *Selected Areas in Communications, IEEE Journal on*, vol. 23, no. 10, pp. 2069–2084, 2005.

[2] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[3] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 97–102.

[4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*. IEEE, 1999, pp. 17–31.

[5] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 113–126.

[6] "Floodlight," 2015. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[7] T. Javid, T. Riaz, and A. Rasheed, "A layer2 firewall for software defined network," in *Information Assurance and Cyber Security (CIACS), 2014 Conference on*. IEEE, 2014, pp. 39–42.

[8] M. Suh, S. H. Park, B. Lee, and S. Yang, "Building firewall over the software-defined network controller," in *Advanced Communication Technology (ICACT), 2014 16th International Conference on*. IEEE, 2014, pp. 744–748.

[9] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks." in *NSDI*, 2013, pp. 1–13.

[10] 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 *Proc. USENIX NSDI*, 2014.

[11] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks." 2013.

[12] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 15–27.

[13] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu, "IPSec/VPN security policy: Correctness, conflict detection, and resolution," in *Policies for Distributed Systems and Networks*. Springer, 2001, pp. 39–56.

[14] E. Lupu and M. Sloman, "Conflict analysis for management policies," in *Integrated Network Management V*. Springer, 1997, pp. 430–443.

[15] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *Software Engineering, IEEE Transactions on*, vol. 25, no. 6, pp. 852–869, 1999.

[16] D. Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2001, pp. 827–835.

[17] E. S. Al-Shaer and H. H. Hamed, "Firewall policy advisor for anomaly discovery and rule editing," in *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*. IEEE, 2003, pp. 17–30.

[18] D. R. Morrison, "Patricia - Practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[19] K. Poornaselvan, S. Suresh, C. D. Preya, and C. Gayathri, "Efficient IP lookup algorithm," *Strengthening the Role of ICT in Development*, p. 111, 2007.

[20] S. Natarajan, X. Huang, and T. Wolf, "Efficient conflict detection in flow-based virtualized networks," in *Computing, Networking and Communications (ICNC), 2012 International Conference on*. IEEE, 2012, pp. 690–696.

[21] P. Gupta and N. McKeown, "Algorithms for packet classification," *Network, iEEE*, vol. 15, no. 2, pp. 24–32, 2001.

[22] F. B. Schneider, "Least privilege and more," in *Computer Systems*. Springer, 2004, pp. 253–258.

[23] "OpenDaylight," 2010. [Online]. Available: https://www.opendaylight.org/