Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments

Sandeep Pisharody, *Student Member, IEEE,*
Janakarajan Natarajan, Ankur Chowdhary, Abdullah Alshalan, *Student Member, IEEE,*
and Dijiang Huang, *Senior Member, IEEE*

Abstract—The ease of programmability in Software-Defined Networking (SDN) makes it 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. However, implementing security solutions in such an environment is fraught with policy conflicts and consistency issues with the hardness of this problem being affected by the distribution scheme for the SDN controllers. In this paper we present Brew, a security policy analysis framework implemented on an OpenDaylight SDN controller, that has comprehensive conflict detection and resolution modules to ensure that no two flow rules in a distributed SDN-based cloud environment have conflicts at any layer; thereby assuring consistent conflict-free security policy implementation and preventing information leakage. We present techniques for global prioritization of flow rules in a decentralized environment, extend firewall rule conflict classification from a traditional environment to SDN flow rule conflicts by recognizing and classifying conflicts stemming from cross-layer conflicts and provide strategies for unassisted resolution of these conflicts. Alternately, if administrator input is desired to resolve conflicts, a novel visualization scheme is implemented to help the administrators view the conflicts graphically. We demonstrate the correctness, feasibility and scalability of our framework through a proof-of-concept prototype.

Index Terms—Software-Defined Networks, Network security, Flow rule conflicts, Distributed environments, Data center network.

1 Introduction

SOFTWARE Defined Networking (SDN) is a transformative approach to network design and implementation, based on the premise of separating the control of network functions from the network devices themselves (switches, routers, firewalls, load balances, etc.). Using the OpenFlow [1] 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, allowing it to satisfy functionalities traditionally fulfilled by a multitude of physical devices. Along with the SDN support of programmable network interfaces, this flexibility makes SDN an ideal platform for multi-tenant data center deployments that require flexibility and dynamism. This is especially true in an Infrastructure-as-a-service (IaaS) cloud where Virtual Machines (VMs) are managed by tenants seeking technological and financial flexibility.

The decoupling of data and control planes in SDN brings about scalability concerns owing to potential bottlenecks at the controller. Studies suggest that although a centralized controller can scale for a respectable enterprise network, it would fail for a data center deployment [2], [3]. While researchers have explored architectures for decentralizing the SDN architecture [4], [5] they do not address flow rule management across this environment.

The flexibility and programmability of SDN allows for the ability to respond rapidly to changing user and security requirements and empowers users in a shared tenant environment to secure their own logical infrastructure in a perceivably private manner. Any security implementation by the tenant such as Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), Deep Packet Inspection (DPI), Virtual Private Networks (VPN), Moving Target Defense (MTD) etc.; would be accomplished by installing new flow rules in the SDN-based cloud environment. However, the shared control plane leaves open the potential for conflicts between flow rules from different tenants. Unlike traditional environments where new rules can get added only through an administrator, abstraction of the data plane from the control plane leads to applications being able to introduce new flow rules into the controller through an API. When done without understanding existing flow rules, the desired security policy or in an adversarial manner, this could result in potential conflicts as well. In a decentralized SDN-based cloud environment with multiple controllers, the policy conflict issue is amplified since conflicts could arise due to different controllers not being in sync, and not having the same view of the environment. To complicate matters further, a dynamically changing network topology adds its own wrinkles.

Just as firewall conflicts in a traditional network limits effectiveness of a security infrastructure [6], conflicts between flow rules on the controller limits the effectiveness and impact of a security implementation in an SDN-based cloud environment. Amongst issues that are heightened in an SDN-based cloud environment are issues caused by flow rule chaining, cross-layer policy conflicts, partial matches and by `set-field` actions.

Substantial research has attempted to address the prob-

- *S. Pisharody, J. Natarajan, A. Chowdhary, A. Alshalan and D. Huang are with the School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287.*
  E-mail: {spishar1, jnatara1, achaud16, aalshala, dhuang8}@asu.edu
- *A. Alshalan is with King Saud University.*

lems brought forth above, significant amongst which are FortNOX [7] and the FlowGuard [8] framework. While they deal effectively with direct flow violations, they do not tackle conflicts across addresses over multiple layers. Consider a multi-tenant SDN-based environment. Often, tenants use flat layer-2 topologies due to latency concerns, and the ability to conduct inline promiscuous monitoring using layer-2 devices [9]. A natural extension would be to implement layer-2 flow rule policies. The data center itself might operate with flow rules based on layer-3 addresses. If different policy enforcement points enforce policies based on different layers, inconsistent actions could result. Cloud environments deployed on an OpenStack environment use OpenFlow based vSwitches on each of the compute nodes for networking. The OpenStack interface that handles all networking `br-tun` uses purely layer-2 addresses, while the gateway that connects the environments externally `br-ex` uses layer-3 addresses, once again causing potential issues. Conflicts across multiple layer addresses or *cross-layer conflicts* become severe in an SDN setup where each SDN switch, both physical and virtual, can be considered to be a distributed firewall instance, each with a different local view of the environment and policy.

In our work, we classify all potential conflicts in an SDN-based cloud environment and including cross-layer conflicts [10]. We develop a methodology and implement a controller-based algorithm for extracting flow rules in a distributed controller environment, and detect intra- and inter-table flow rule conflicts utilizing cross-layer conflict checking. Further, we address automatic and assisted conflict resolution mechanisms and present a novel visualization scheme for conflict representation. This work is implemented in a security policy analysis framework named Brew, built on a OpenDaylight (ODL) based SDN controller, that effectively scrubs the flow table in a distributed SDN-based cloud environment, highlights and resolves potential conflicts. To summarize, Brew

- Includes techniques for global prioritization of flow rules in a decentralized environment depending on the decentralization strategy.
- Extends firewall rule conflict classification in a traditional environment to SDN flow rule conflicts by identifying cross-layer conflicts, (which tend to be temporal in nature).
- Detects flow rule conflicts in a multiple, decentralized controller based SDN-based cloud environments including conflicts between flow rules implementing QoS requirements.
- Provides strategies for unassisted resolution of these conflicts.
- Presents a novel visualization scheme, implemented to help the administrators view flow rule conflicts graphically.

This paper is organized as follows. In Section 2, we present some related work. In Section 3 we produce real-world motivating scenarios for our framework and present the formalism for flow rule conflicts and classification. Next, we present our system design in Section 4. The implementation specifics are discussed in Section 5, and evaluation information is shared in Section 6. Finally, we show our conclusions and our future research tasks in Section 7.

2 Related Work

While advances in SDN have made it central to deployment of a cloud environment, security mechanisms in SDN trail its applications. Javid et al. [11] built a layer-2 firewall, and Suh et al. [12] illustrated a proof-of-concept version of a traditional layer-3 firewall over an SDN controller. But these works do not address the problem of conflicting flow rules.

VeriFlow [13] 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. Pyretic [14] 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 caused by cross-layer conflicts in a distributed SDN-based cloud environment cannot be handled by Pyretic without a flow tracking mechanism. FRESCO [15] introduces a Security Enforcement Kernel (SEK) and allows security services to provide reusable modules accessible through a Python API. While the SEK prioritizes rules from security applications to address conflicts, it does not tackle indirect security violations, partial violations or cross-layer conflicts.

FortNOX [7] is an extension to the NOX controller that implements role-based and signature based enforcement to ensure applications do not circumvent the existing security policy, thereby enforcing policy compliance and protecting the flow installation mechanism against adversaries. Conflict analysis in FortNOX doesn't consider inter-dependencies within flow tables, and decision making seems to follow a least permissive strategy instead of deciding keeping the holistic nature of the environment in mind. Moreover, it uses only layer-3 and layer-4 information for conflict detection, which is incomplete since SDN flow rules could use purely layer-2 addresses. In addition, FortNOX doesn't appear to be able to handle partial flow rule conflicts or cross-layer conflicts.

FlowGuard [8] 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.

Onix [4] facilitates distributed control in SDN by providing each instance of the distributed controller access to holistic network state information through an API. HyperFlow [5] synchronizes the network state among the distributed controller instances while making them believe that they have control over the entire network. Kandoo [2] is a framework tailored for a hierarchical controller setup. It separates out local applications that can operate using the local state of a switch; and lets the root controller handle applications that require network-wide state. DISCO [16] is a distributed control plane that relies on a per domain organization, and contains an east-west interface that manages communication with other DISCO controllers. It is highly suitable for a hierarchically decentralized SDN controller environment.

ONOS [17] is an OS that runs on multiple servers, each of which acts as the exclusive controller for a subset of switches and is responsible for propagating state changes between the switches it controls. Several of these works serve as essential groundwork for the controller decentralization strategy that is employed in the Brew framework.

The works discussed above do not tackle cross-layer policy conflicts in distributed environments, or address administrator assistance free conflict resolution, which is very important in dynamic SDN-based cloud environments. To that end, our previous work [10] proposed a new classification type to describe cross-layer conflicts. Using the formalism provided by this new type of flow rule conflict, we detect and introduce mechanisms to resolve the conflicts without administrator assistance.

One of the earliest work in visualization of rule conflicts has been done by Mayer et al. [18] in Fang, where the administrator can view firewall rules in simple text format, devoid of any graphics. Similarly, in Firewall Policy Advisor [6], Al-Shaer and Hamed display simplified versions of complex firewall rules, and show firewall rule conflicts in tabular format. PolicyVis [19] used overlapping bars to represent conflict types, and colors to represent the action. However, the conflicts are visible only when a certain scope is defined. In FAME [20], Hu et al. used a matrix to represent conflicting and non-conflicting address segments; but fails while trying to represent larger rule sets. A sunburst visualization is used by Mansmann et al. [21] to visualize the rule set, but does not provide any visualization for flow rule conflicts. None of these works, however provide scalable rule conflict visualization to the administrator, providing high level conflict categorization, with granular information provided upon need.

3 Background & Models

In this section, we cover some underlying information including formal definitions of a flow rule, flow rule management challenges and discuss scenarios motivating this work.

3.1 Motivating Scenario

One of the major benefits of using SDN to implement a cloud environment is the ability to have multiple applications run on the SDN controller, each of which has complete knowledge of the cloud environment. This can be leveraged by the cloud provider to provide Security-as-a-Service (SaaS). A few potential examples of services in a SaaS suite are Firewalls, VPN, IDS, IPS, MTD etc. Implementing a management system that only specifies security policies without tackling topological interaction amongst constituent members has always been a recipe for conflicts [22].

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 policy inconsistencies caused by *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-based cloud environments could be.

We discuss three distinct case studies in an SDN-based cloud environment where the security of the environment is put at risk due to flow rule conflicts. The first scenario serves as an example where rules from different applications conflict with each other, and the second scenario serves as an example where rules from a single module might cause conflicts due to the dynamism in the environment.

3.1.1 *Case Study 1: VPN Services*

In a multi-tenant hosted data center, 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. If an application on a different controller inserts policies to implement 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. Given such a scenario, our work detects and determines the flow rule conflicts and helps resolve them to keep the flow table manageable and up-to-date.

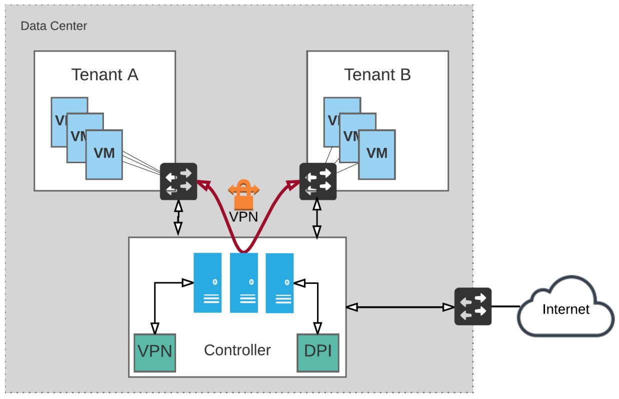


Fig. 1. Policy conflicts caused by different applications in an SDN-based cloud.

3.1.2 *Case Study 2: Moving Target Defense (MTD)*

Moving Target Defense (MTD) techniques have been devised as a tactic wherein security of a cloud environment is enhanced by having a rapidly evolving system with a variable attack surface; giving defenders an inherent information advantage [23]. An effective countermeasure used in MTD is network address switching, which can be accomplished in SDN with great ease. Since an MTD application could dynamically and rapidly inject new flow rules into an environment, it could lead to the very problem we are trying to address.

In the data center network shown in Figure 2, we have Tenant A hosting a web farm. Being security conscious, only traffic on TCP port 443 is allowed into the IP addresses that belong to the web servers. When an attack directed against host $A2$ has been detected, the MTD application responds with countermeasures and takes two actions: *a*) a new webserver (host $A3$) is spawned to handle the load of host $A2$; and *b*) the IP for host $A2$ is migrated to the Honeypot network and assigned to host Z.

In order to run forensics, isolate and incapacitate the attacker, the HoneyPot network permits all inbound traffic, but no traffic outward to other sections of the data center. These actions result in new flow rules being injected into the flow table that *a*) permits *all* traffic inbound to the IP that originally belonged to host $A2$, but now belongs to host Z; *b*) modifies an incoming packet's destination address from host $A2$ to host $A3$ if the source is non-adversarial; *c*) stops all outbound traffic from the IP that originally belonged to host $A2$, but now belongs to host Z to the rest of the data center; and *d*) permits traffic on port 443 to host $A3$ (not of great importance to our case). The original policy allowing only port 443 to the IP of host $A2$, and the new policy allowing all traffic to the IP address of host Z are now in conflict. Our earlier work [23] uses effective policy checking in SDN-based MTD solutions in cloud deployments.

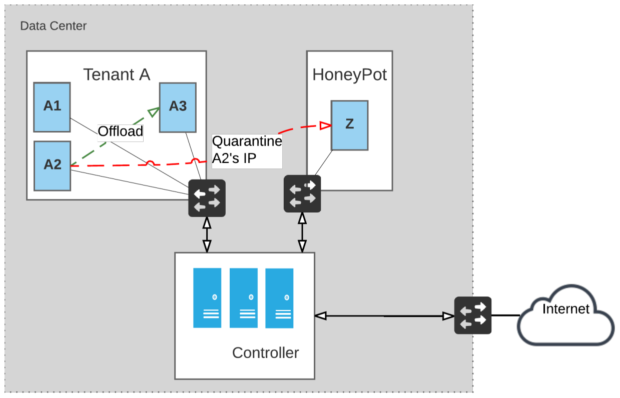


Fig. 2. Policy conflicts in SDN-based cloud caused by MTD.

3.1.3 *Case Study 3: Load Balancing & Intrusion Detection*

Like the scenario in Case Study #1, consider an SDN-based data center environment where a load balancing application as well as an IDS application run on the SDN controller. Upon detecting intrusions, the IDS could implement a countermeasure that offloads traffic from the compromised node. However, the load balancing application which routes new connections based on their active load might start redirecting new traffic to the compromised node, since the system would infer that the compromised node has the least amount of load.

3.2 Flow Rule Model

To formally create a model that describes flow rules in an SDN-based cloud environment, we first define an address n.

Definition 1. *An address n is a 6-tuple representing the address space $(\epsilon_s, \epsilon_d, \zeta_s, \zeta_d, \eta_s, \eta_d)$, where ϵ represents the OSI layer-2, ζ represents layer-3, and η represents layer-4 addresses; with subscript s denoting source and d denoting destination addresses.*

If we let N be the entire range of addresses in the 6-tuple address space, then we have:

Definition 2. *A flow rule r is a transform function $f : N \to N$ that transforms n to n', where n' is $(\epsilon'_s, \epsilon'_d, \zeta'_s, \zeta'_d, \eta'_s, \eta'_d)$ together with an associated action set a. Thus,*

$$r := f(n) \rightsquigarrow a$$

Definition 3. *An action set a is the set of all associated actions taken by the controller on a flow match. Atomic actions include: a) forwarding to specific ports on the switch; b) flooding the packet on all ports; c) changing QoS; d) encapsulating e) encrypting; f) rate limiting; g) drop the packet; and h) customizable actions using various `set-field` actions.*

The `set-field` capabilities in the action fields of the rules ensures that any, all or none of the fields in n may be modified as a result of the transform function r. Considering cases where the action set a is a pointer to a different flow table, we can apply the transform function on the result of the original transform function n'. Formally, if $r := f(n) \rightsquigarrow a$; $f(n) = n'$ and $a := g(n') \rightsquigarrow a'$ then,

$$r := g(f(n)) \rightsquigarrow a'$$

Thus, multiple rules applied in succession to the same input address space can simply be modeled as a composite function. It must be noted that the complexity of the flow rule composition function would be exponential in nature, since each flow rule could have multiple actions, each of which themselves could recursively lead to multiple atomic actions.

In a flow table R containing rule set $\{r_1, r_2, ..., r_n\}$, with each flow rule $r_i \in R$ being a 6-tuple $(p_i, \alpha_i, \nu_i, n_i, \rho_i, a_i)$, we have *a*) p as the priority of the rule defined in the range $[1, 65535]$; *b*) α is the ingress port; *c*) ν is a VLAN ID; *d*) n is the 6-tuple address space defined earlier; *e*) ρ as the layer-4 protocol; and *f*) a as the action set for the rule. Flow rules also contain packet counters and timeout values, but they are not relevant match or action fields in rule processing. Match fields α and ν, representing ingress port and VLAN ID merely eliminate and not add to potential conflicts. Hence, we do not include them in further discussion. Of these fields, we focus on just priority, match fields and instructions to build the flow rule conflict problem. Table 1 shows sample flow table rules. Note that the action field in the table has been modified to a simplistic forward and drop, as opposed to their implementation in OpenFlow, which specifies which the egress port is, and whether the packet is to be broadcast etc.

3.3 Security Policies using Flow Rules

Due to the ability to alter headers from multiple layers of the OSI stack, flow rules in the OpenFlow protocol can inherently be used for traffic forwarding, routing and traffic shaping. Research has shown that, in addition to traffic manipulation functionalities, most security policies can be transferred into flow entries and deployed on OpenFlow devices [24].

While several security mechanisms implemented in traditional environments depend on routing traffic through middleboxes [25], it has been demonstrated that integrating processing into the network is just as effective [26]. The centralized control in the SDN paradigm makes can make this integration simple and elegant. Models to implement traditional security functions such as firewall rules, Intrusion Detection System (IDS) and Network Address Translation (NAT) rules in software have been demonstrated to be successful [27]. SIMPLE, a framework that achieves OpenFlow based enforcement of middlebox policies has been demonstrated in [28]. Further, a multi-level security system using OpenFlow that implements desired security policies using flow rules to accomplish network traffic monitoring as well as verification of packet contents has also been successfully implemented [29]. Authors in [30] survey these and several other security implementations using OpenFlow.

A typical firewall rule, that blocks all Telnet traffic can be specified in OpenFlow as (ignoring all irrelevant fields):

```
Protocol = TCP
TCP destination port = 23
actions: DROP
```

Similarly, a load balancer policy, IPS/IDS policy or a NAT policy could be implemented by modifying the layer-3 source or destination address to send the flow to a specific device as follows:

```
SRC IP = 10.5.50.5
DST IP = 10.211.1.1
actions: DST IP = 10.211.1.63, FORWARD
```

In this paper, we assume that security policies being implemented, such as packet filtering rules, IDS and IPS rules use the flow rule tuples detailed in Section 3.2.

3.4 Flow Rule Management Challenges

Switches in an SDN-based cloud environment maintain at least one flow table, consisting of match conditions and associated actions. An ingress packet is matched against the flow table entries to select the entry that best matches the ingress packet, and the associated instruction set is executed. Such an instruction may explicitly direct the packet to another flow table, where the same process is repeated. When processing stops, the packet is processed with its associated action set. Unlike traditional firewalls which process matches based on a *first match* approach, which selects the first rule that matches the address space of the incoming packet, SDN generally uses a Most Specific Take Precedence (MSTP) approach, where the rule with the most specific match for the incoming flow is preferred [1].

Since flow rules can match more than just layer-3 and layer-4 headers as in a traditional network, they are inherently more complex by having additional variables to consider for a match. 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 9. This flow would clearly match rule 8 in Table 1 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 3.5). Since these conflicts arise because of addressing across different OSI layers, we categorize them differently.

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, ensuring synchronization of the flow rules on all the distributed controllers is challenging. Additionally, ensuring that the changing flow rules are always in line with the security policy of the organization is not trivial.

3.5 Flow Rule 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 4. *An address space $\xi_i \subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{s_i} \subseteq \xi_{s_j} \wedge \xi_{d_i} \subseteq \xi_{d_j}$.*

Definition 5. *An address space $\xi_i \not\subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{s_i} \not\subseteq \xi_{s_j} \vee \xi_{d_i} \not\subseteq \xi_{d_j}$.*

Definition 6. *An address space $\xi_i \subset \xi_j$ if and only if they refer to the same OSI layer, and $(\xi_{s_i} \subset \xi_{s_j} \wedge \xi_{d_i} \subseteq \xi_{d_j}) \vee (\xi_{s_i} \subseteq \xi_{s_j} \wedge \xi_{d_i} \subset \xi_{d_j})$.*

Definition 7. *Address space intersection $\xi_i \cap \xi_j$ produces a tuple $(\xi_{s_i} \cap \xi_{s_j}, \xi_{d_i} \cap \xi_{d_j})$ if and only if ξ_i and ξ_j refer to the same OSI layer.*

Definition 8. *Conflict detection problem [31] seeks to find rules r_i, r_j such that $r_i, r_j \in R$ and $(n_i \cap n_j \neq \emptyset) \wedge (\rho_i = \rho_j) \wedge (a_i \neq a_j \vee p_i \neq p_j)$.*

Since flow rules in an SDN-based cloud environment are clearly a super-set of rules in a traditional firewall environment, work on flow rule conflicts are an extension of the work on firewall rule conflicts. While several works have classified firewall rule conflicts [31], [32], [33]; the seminal work by Al-Shaer and Hamed [6] is often used to classify firewall rule conflicts in a single firewall environment. Our previous work [10] built on that work and introduced a new classification of conflicts that better describes conflicts between address space overlap over multiple OSI layers.

Knowing that OpenFlow specifications clarify that if a

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	*	*	forward
2	50	*	*	10.5.50.5	10.211.1.63	tcp	*	80	forward
3	52	*	*	10.5.50.5	10.211.1.0/24	tcp	*	*	forward
4	53	*	*	10.5.50.0/24	10.211.1.63	tcp	*	*	drop
5	54	*	*	10.5.50.5	10.211.1.63	tcp	*	*	drop
6	51	*	*	10.5.50.0/16	10.211.1.63	tcp	*	*	drop
7	55	*	*	10.5.50.5	10.211.1.0/24	tcp	*	80-90	drop
8	57	11:11:11:11:11:ab	11:11:aa:aa:11:11	*	*	*	*	*	forward
9	58	*	*	*	*	tcp	*	80	drop

TABLE 1
Flow table example.

packet matches two flow rules, only the flow rule with the highest priority is invoked, conflicts in SDN flow rules can be classified into:

- **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 1 has a address space that is a subset to the address space of rule 1, with matching protocol and actions. Hence, rule 2 is redundant to rule 1. Redundancy 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 [6]. For example, rule 4 in Table 1 has the same address 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 address space of rule r_j is matched by rule r_i [6]. As shown in Table 1, rule 1 is a generalization of rule 5, since the address space of rule 5 is a subset of the address 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*) address space $\epsilon_i \nsubseteq \epsilon_j \wedge \zeta_i \nsubseteq \zeta_j \wedge \eta_i \nsubseteq \eta_j \wedge \epsilon_i \nsupseteq \epsilon_j \wedge \zeta_i \nsupseteq \zeta_j \wedge \eta_i \nsupseteq \eta_j \wedge (\epsilon_i \cap \epsilon_j \neq \emptyset \vee \zeta_i \cap \zeta_j \neq \emptyset \vee \eta_i \cap \eta_j \neq \emptyset)$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i \neq a_j$. As shown in Table 1, rule 3 is correlated to rule 4. Since multiple SDN flow rules can have the same priority, the following condition also leads to a correlation conflict [10]: *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 1, rule 6 is correlated to rule 1.
- **Overlap:** A rule r_i overlaps rule r_j iff *a*) address space $\epsilon_i \nsubseteq \epsilon_j \wedge \zeta_i \nsubseteq \zeta_j \wedge \eta_i \nsubseteq \eta_j \wedge \epsilon_i \nsupseteq \epsilon_j \wedge \zeta_i \nsupseteq \zeta_j \wedge \eta_i \nsupseteq \eta_j \wedge (\epsilon_i \cap \epsilon_j \neq \emptyset \vee \zeta_i \cap \zeta_j \neq \emptyset \vee \eta_i \cap \eta_j \neq \emptyset)$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i = a_j$. An overlap rule is the complementary conflict to a correlation; but with the flow rules in question having matching action sets instead of opposing actions. This overlap can be seen between rule 6 and rule 7 in Table 1.
- **Imbrication:** Flow rules where not all OSI addresses layers have match conditions could result in cases where, *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). Address space overlaps between such rules are classified as imbrication [10]. Imbrication conflicts are more complex than the other conflict classifications, since they are temporal in nature. For example, the mapping between a layer-2 MAC address and layer-3 IP addresses might result in a conflict between two flow rules at time t_1 in the layer-3 address space. But if the IP-MAC address mapping changes, there may not be an address space overlap between the two rules at time t_2. This makes imbrication conflicts hard to find and resolve [10]. Using the topology shown in Figure 9 and the flow rules in Table 1, it can be seen that flow rule 4, which denies traffic from host A to host C and flow rule 8, which permits traffic from host A to host C are clearly imbricates.

3.6 Controller Decentralization Model

Centralizing the control plane in SDN is fraught with scalability challenges associated with the SDN controller being a bottleneck [2]. While benchmarking tests on an SDN indicate that successful processing of about $30,000$ requests per second [34], it does not scale well enough to be deployed in a cloud environment [3]. Distributing the controller responsibilities to multiple devices is an obvious solution.

Choosing a decentralized control architecture is not trivial. There are several controller placement solutions, and factors such as the number of controllers, their location, and communication topology impact network performance.

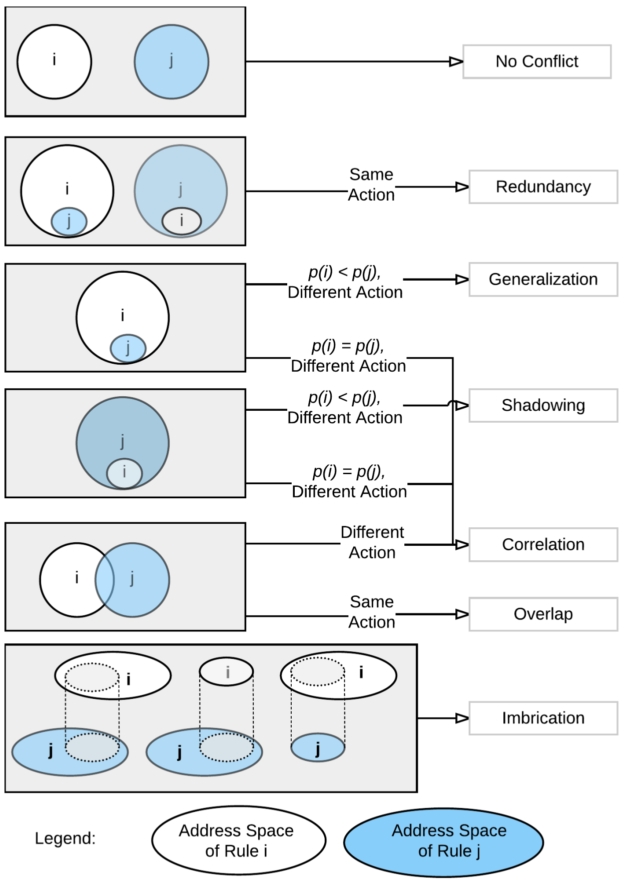


Fig. 3. Venn Diagram showing address space overlap and flow rule conflicts.

Three major issues need to be addressed in determining the decentralization architecture: *a*) efficient east and west-bound APIs need to be developed for communication between SDN controllers; *b*) the latency increase introduced due to network information exchange between the controllers need to be kept to a minimum; and *c*) the size and operation of the controller back-end database needs to be evaluated.

Since the key piece of information required for accurate flow rule conflict detection and resolution is the priority value p, the key challenge in extending flow rule conflict resolution from a single controller to a distributed SDN-based cloud environment lies in associating global priority values to flow rules. The strategies to associate these global priority numbers to flow rules in different decentralization scenarios differ drastically. We classify four different multiple controller scenarios, and the global priority assignment logic followed by our framework for each of them.

3.6.1 Host Partitioning

This partitioning method is most like a traditional layered network architecture, where an SDN controller now handles the functionalities of an access level switch, combined with the intelligence of a router and access control server. The SDN-based cloud environment is separated into domains, where each domain is controlled by a single controller. The controllers then communicate with each other using east-west communication APIs. Running on the assumption that the controller knows best about the nodes it is responsible for, flow rules which contain match conditions with addresses *local* to the controller are preferred. Flow rule conflict resolution in this scenario doesn't need to consider every flow rule in the environment, but only ones for the local domain. While host partitioning is the most intuitive decentralization strategy, it eliminates some of the flexibility provided by the network topology dynamicity in SDN.

3.6.2 Hierarchical Controllers

Hierarchical controller architecture is a variant of host partitioning. The difference lies in assigning the priority of the flow rules. Instead of the local flow rules having the highest priority, in a hierarchical controller architecture, the local (or leaf) controllers have the lowest priority. Moreover, the partitioning is not strictly host based, as administrators could decide to run certain applications at a leaf level, and certain applications at higher level controllers. For example, a DHCP application could reside on the leaf controller; while a NAT application could reside on the root controller. In case of conflicts, the flow rules originating from applications on the root controller are preferred.

3.6.3 Application Partitioning

For a multiple SDN controller environment where each controller handles specific applications or application groups, associating a priority values is straightforward. By assigning a weight to each application [7] the global priorities of flow rules generated by all applications can be determined. For example, consider Controller A has security applications running on it, and Controller B has QoS and traffic shaping applications running on it. If security applications are prioritized with a higher weight than traffic shaping applications, two flow rules with the same priority generated by applications on Controller A and Controller B will end up with the rule generated by Controller B having a lower global priority. An alternate strategy to assign global priority values would be to allocate ranges for flow rules created by applications. For example, it could be decided that any NAT rule generated by the NAT application on the controller must be within a priority of $40,000$ and $42,000$. Thus a global priority for a NAT rule would be generated by mapping the priority originally in the range $[1, 65535]$ to a global priority in the range $[40000, 42000]$.

3.6.4 Heterogeneous Partitioning

In heterogeneous decentralized environment, appealing aspects of each of the above decentralization scenarios are combined to obtain the optimal situation for meeting the requirements. Careful consideration needs to be taken to identify the priorities of applications and controllers before deployment, to have a conflict resolution strategy. Alternately, the conflict resolution could be reverted to a manual process where administrators provide the conflict resolution decision.

3.7 Conflict Resolution Model

The different flow rule conflicts can be broadly categorized into Intelligible and Interpretative conflicts. The resolution strategies for each of these two categories are markedly different, and are detailed in the remainder of this Section.

3.7.1 *Intelligible conflicts*

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 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 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.

3.7.2 *Interpretative conflicts*

Conflicts that fall into Generalization, Correlation and Imbrication classification cannot be intuitively resolved without any loss of 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 per the cloud environment in question. Removing these conflicts is a complex problem [35]. We discuss a few resolution strategies that could be applied to resolving these conflicts below. We use a generic quantifier called the Conflict Resolution Criteria (CRC) which is dependent on the resolution criteria in use. A data structure to be used as a concomitant to the flow rules refers to this CRC metric, as shown in Figure 5.

- 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 the most popular strategies in conflict resolution.
- 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 [7]. Thus, a flow rule originating from a security module is prioritized over flow rule from an application or optimization module. Table 2 shows sample precedence and associated CRC values for a few generic applications that might run in an SDN-based cloud.
- Environment calibrated - This strategy incorporates learning strategies in the environment to make an educated decision on which conflicting flow rule 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 of 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. An alternate environment calibrated approach might involve quantitative security analysis of the environment with each of the conflicting rules, and picking the safest option.
- 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. Visual assistance tools incorporated as part of the Brew framework assist the administrator decide.

Application	Precedence	CRC
Deep Packet Inspection	2	250
Network Address Translation	3	200
Virtual Private Network	1	255
Quality of Service	4	100
Domain Name Service	5	1

TABLE 2
Example security precedence.

4 System Architecture

4.1 System Modules

Our framework Brew, as shown in Figure 4 is an intuitive model to help resolve conflicts in flow rules in a distributed SDN-based cloud environment. It consists of two inter-related modules, OFAnalyzer and OFProcessor, that together achieve a conflict free flow table. These modules all operate at the control plane level i.e. their operations are uninhibited by either the physical topology or the logical topology as seen by the different tenants. Brew runs as an application on the controller that listens for new/modified flow rules being introduced into the system as part of the OFAnalyzer module. The processing is broadly compartmentalized to sanitization, conflict detection and conflict resolution, as part of the OFProcessor module. The modules that accomplish these tasks are detailed in the remainder of this Section.

4.1.1 *Flow Extraction Engine*

The flow extraction engine functions as part of the OFAnalyzer module in Brew. It intercepts any new or updated flow rule that is being injected into the controller from different modules. 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 as discussed in Section 3.6. Thus, the priority assigned by the flow extraction engine may differ from the priority of the

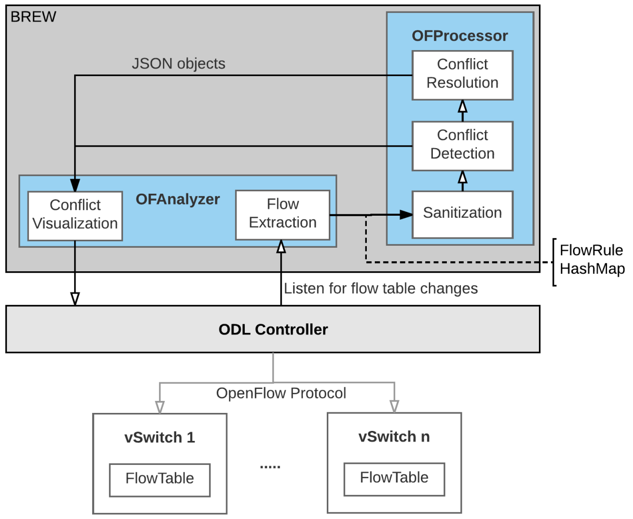


Fig. 4. System overview representing different Brew modules.

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.

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 extracted OpenFlow rule using four additional fields over 32 bits of information as shown in Figure 5. 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 CRC metric (discussed in Section 3.7). Armed with these additional bits of information, detection of flow rule conflicts using the methodology shown in Algorithm 1 is now possible.

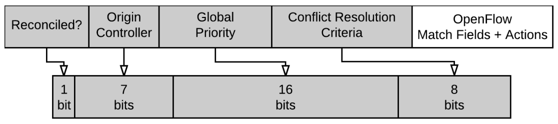


Fig. 5. Data structure format.

4.1.2 Sanitization Engine

The sanitization engine functions as part of the OFProcessor module in Brew. 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, deny and QoS remain. The atomization process itself follows along the lines of `iptables` processing in Unix with modifications based on the formal model described in Section 3. Since QoS and packet counters can be processed along with the permit and deny actions, flow rules with QoS and traffic engineering actions are not processed any further. 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.

Flow rules which have only layer-2 address spaces in its match conditions are next mapped 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. 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.

4.1.3 Conflict Detection Engine

The conflict detection engine functions as part of the OFProcessor module in Brew. Once the flows present in the system has been extracted and processed, the conflict detection engine identifies and classifies conflicts based on the categories described in Section 3.5.

Determining the existence of address space overlap between flow rules is the first step in deciding if a conflict exists between two flow rules. The address space overlap is detected using a Patricia trie lookup [36] based algorithm. The Patricia trie is an efficient search structure for finding matching IP strings with a good balance between running time (lookup and update) and memory space requirement, and has been used previously with great success [37]. The layer-3 address space for each rule is stored in eight different Patricia trie data structures. Each edge in the Patricia trie is labeled with a bit, with each leaf node corresponding to the stored string. The semantic information is preserved by having the leaf nodes store the unique identifier for every flow rule that has matches that octet. We do an octet wise Patricia trie lookup 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. An address space overlap is determined if two rules share the leaf node on all eight Patricia tries. Wildcard matches are handled by creating a superset that includes all rules in the child nodes' leaves. Once an address space overlap is determined, evaluating if a conflict exists between the flow rules can be accomplished in constant time using simple comparison operations.

4.1.4 Conflict Resolution Engine

Once the flow rule conflicts have been detected, the conflict resolution module is invoked. Intelligible conflicts are resolved automatically. However, in case of interpretative conflicts (Generalization, Correlation and Imbrication) which cannot be resolved without loss of information, the CRC metric is used to determine the resolution strategy (discussed in Section 3.7) to be employed. Note that the CRC metric is used only while resolving interpretative conflicts.

5 Implementation

Brew was implemented on an OpenDaylight (ODL) SDN controller. ODL is an open-source project under The Linux Foundation, and has REST APIs for use by external applications. Our work extends the stable version (Lithium) of the ODL controller. By modularizing the functionality of Brew into the front-end OFAnalyzer interface, and a back-end OFProcessor, as shown in Figure 4; we hope to have the same OFProcessor back-end work with multiple SDN controller specific OFAnalyzer interfaces in the future.

5.1 OFAnalyzer

The OFAnalyzer module acts as the interface between the SDN controller and the OFProcessor back-end. It performs two important tasks, *a*) flow extraction; and *b*) conflict visualization. The flow extraction engine in the OFAnalyzer listens and extracts flow rules from two different datastores maintained by ODL, as shown in Figure 6. Classified broadly on the type of data maintained in them, they are: *a*) configuration datastore; and *b*) operational datastore. The configuration datastore on each ODL controller contains data that describe the changes to be made to the flow rules on the switches. Candidate flow rules sent by all applications reside in this tree before they are sent to the devices. The operational datastore matches the configuration datastore in structure, but contains information that the controller discovers about the network through periodic queries. Listening to flow rules from both datastores helps the OFAnalyzer maintain a complete view of the flow rules present in the environment, especially in a distributed controller scenario. The source of the rules is noted to eliminate duplication.

OFAnalyzer listens for changes in the flows present in

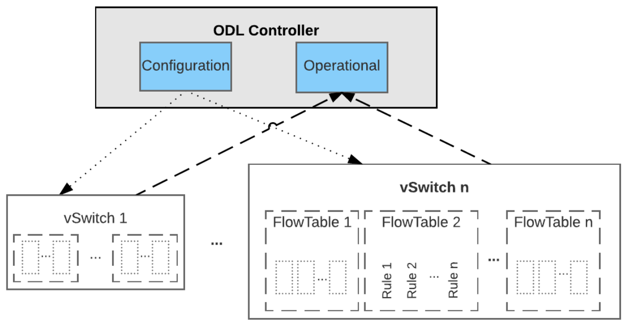


Fig. 6. ODL datastores.

the operational datastore by registering itself as a listener using a `DataBroker` Document Object Model (DOM) tree to be informed of any addition or modification to flows. The flow rules and the associated conflict information (if any) are obtained using a REST Request. In addition, the flow extraction engine also listens for candidate flow rules from different applications running on the controllers. The results, obtained as a list of JSON objects are prepared for visualization using JavaScript conversion routines. Multiple visualization schemes then display this information to the administrator in a manner of his/her choosing.

The configuration datastore is handled simply by placing OFAnalyzer as sort of a redirect - extracting candidate flow rules and piping it through the OFProcessor prior to any new flow rules being placed in the configuration datastore. The required data is received using a Java Data Transfer Object (DTO). OFAnalyzer extracts the flows from the DTOs and stores it in a local HashMap. Once the flows have been extracted, each flow is given a unique identifier, making it easier to track the flow when analyzing conflicts, and for visualization purposes. The HashMap is then passed to the OFProcessor.

Upon receiving conflict information back from the OFProcessor, the visualization engine details and displays this information in a manner that is both intuitive and concise. The conflict visualization engine is implemented as a module under the DLUX user interface. A hierarchical edge bundling [38] is used to represent the rule relationships using the `D3.js` JavaScript library. This scheme highlights the overall relationship between all the flow entries while simultaneously reducing clutter. Figure 7 shows an example of the hierarchical edge bundling structure showing conflicts in a flow table, with the color of link distinguishing between the relationship between the rules. Details on the conflict between the rules are provided using the Reingold-Tilford tree [39] that presents the details in an aesthetically pleasing and tidy fashion. Figure 8 shows a screenshot of an interactive Reingold-Tilford tree showing conflicts for a single flow rule. Unlike the current rule conflict visualization implementations, our approach focuses on providing a visual interface for the administrator to quickly determine where the most damage causing conflicts are so he/she can resolve it.

5.2 OFProcessor

The OFProcessor module handles the back-end logic of Brew. Its functionalities are compartmentalized as sanitization, conflict detection and conflict resolution. First, the sanitization process obtains the flow rules from the OFAnalyzer and assigns a global priority to the flow based on the decentralization strategy discussed in Section 3.6. The flow rules are then atomized and the `reconciled` bit in the data structure shown in Figure 5 is determined.

The Patricia trie data structure is then used as shown in Algorithm 1 to determine and classify conflicts. 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. All detected conflicts are classified as shown in Figure 3. Since we formally describe any overlaps involving reconciled rules as imbrication conflict, we process those rules separately from non-reconciled rules and classify them as such. The results are stored in a HashMap,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2726066, IEEE Transactions on Dependable and Secure Computing

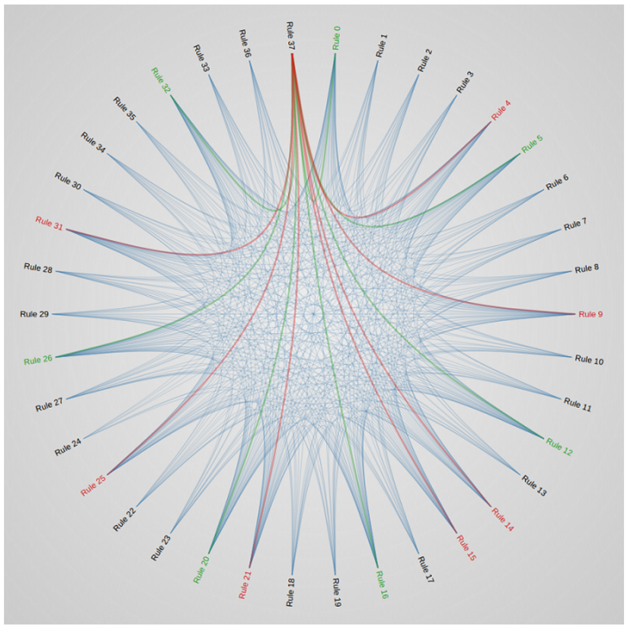
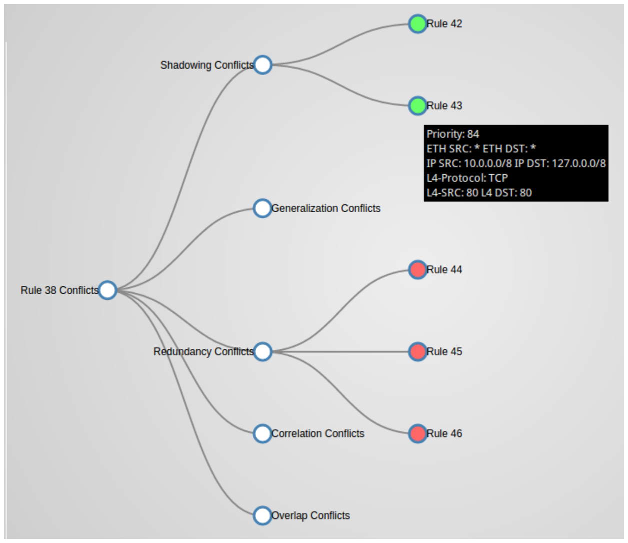


Fig. 7. Hierarchical edge bundling.



Priority: 84
ETH SRC: * ETH DST: *
IP SRC: 10.0.0.0/8 IP DST: 127.0.0.0/8
L4-Protocol: TCP
L4-SRC: 80 L4 DST: 80

Fig. 8. Reingold-Tilford tree.

Algorithm 1: Conflict Detection Engine

```
Input   : $Rule\ r$, $FlowTable\ f$
Output  : Conflict-free $FlowTable\ f'$
Procedure ConDet()
1   $r \leftarrow$ Sanitize $(r)$
2   if $!r.reconciled$ then
3       $\gamma \leftarrow$ SearchPatricia($r.l3addr$)
4       if $r.protocol\ != \ \gamma.protocol$ then
5           return $AddFlow\ (f, r)$
6       else if $r.addr\ \subseteq\ \gamma.addr$ then
7           if $r.action\ ==\ \gamma.action$ then
8               return $ConRes\ (r, \gamma, f, Redundancy)$
9           else if $r.priority\ ==\ \gamma.priority$ then
10              return $ConRes\ (r, \gamma, f, Correlation)$
11          else if $r.priority\ <\ \gamma.priority$ then
12              return $ConRes\ (r, \gamma, f, Shadowing)$
13      else if $\gamma.addr\ \subseteq\ r.addr$ then
14          if $r.action\ ==\ \gamma.action$ then
15              return $ConRes\ (r, \gamma, f, Redundancy)$
16          else if $r.priority\ ==\ \gamma.priority$ then
17              return $ConRes\ (r, \gamma, f, Correlation)$
18          else if $r.priority\ >\ \gamma.priority$ then
19              return $ConRes\ (r, \gamma, f, Generalization)$
20      else if $r.addr\ \cap\ \gamma.addr\ !=\ \emptyset$ then
21          if $r.action\ ==\ \gamma.action$ then
22              return $ConRes\ (r, \gamma, f, Overlap)$
23          else
24              return $ConRes\ (r, \gamma, f, Correlation)$
25  else
26      for $Rule\ \gamma \in f$ do
27          if $r.protocol\ !=\ \gamma.protocol$ then
28              return $AddFlow\ (f, r)$
29          else if $r.addr\ \cap\ \gamma.addr\ !=\ \emptyset$ then
30              return $ConRes\ (r, \gamma, f, Imbrication)$
31          else
32              return $AddFlow\ (f, r)$
```

which the OFProcessor sends back to the OFAnalyzer for visualization purposes. The conflict information sent uses an encoding scheme using the unique flow rule identifier thereby ensuring scalability. Next, the conflict resolution module is invoked.

Once the conflicts between different flow rules have been detected, the conflict resolution process attempts to resolve these. The intelligible conflicts are resolved trivially and the interpretative conflicts are resolved using the CRC metric. Since resolution of interpretative conflicts is lossy, Brew has a manual mode, where administrator input using the conflict visualization functionality offered in the OFAnalyzer to help guide an informed decision. Visualization aids such as Figure 7 and Figure 8 assist administrators in making an educated decision regarding resolution of interpretative conflicts. By hovering over the rule numbers that populate the perimeter of the circle in Figure 7, all flow rules that conflict that specific rule are highlighted. The color schemes indicate the priority of the conflicting rules. Rules highlighted in green have higher priority than the selected rule, indicating to the administrator that modifying this rule would not affect the others. Rules in red have a lower priority than the selected rule, serving to remind the administrator that any change to this rule would affect packet processing. Clicking on the rule number loads the Reingold-Tilford tree, and hovering over the leaves of the tree would display more details about the rules, so the administrator can now make an informed decision by cross checking with those rules.

Algorithm 2: Conflict Resolution Engine

```
Input  : $Rule\ r$, $Rule\ \gamma$, $FlowTable\ f$, $String\ ConflictType$
Output : Conflict-free $FlowTable\ f'$
Procedure ConRes()
1     if ConflictType == Shadowing || ConflictType == Redundancy then
2         return $f$
3     else if ConflictType == Correlation then
4         if $\gamma.CRC > r.CRC$ then
5             $r.addr \leftarrow r.addr - \gamma.addr$
6             $f' \leftarrow$ AddFlow $(f, r)$
7         else
8             $f' \leftarrow$ RemoveFlow $(f, \gamma)$
9             $\gamma.addr \leftarrow \gamma.addr - r.addr$
10            $f' \leftarrow$ AddFlow $(f, r)$
11            $f' \leftarrow$ AddFlow $(f, \gamma)$
12    else if ConflictType == Generalization then
13        $f' \leftarrow$ RemoveFlow $(f, \gamma)$
14        $\gamma.addr \leftarrow \gamma.addr - r.addr$
15        $f' \leftarrow$ AddFlow $(f, \gamma)$
16        $f' \leftarrow$ AddFlow $(f, r)$
17    else if ConflictType == Overlap then
18        $r.addr \leftarrow r.addr + \gamma.addr$
19        $f' \leftarrow$ RemoveFlow $(f, \gamma)$
20        $f' \leftarrow$ AddFlow $(f, r)$
21    else if ConflictType == Imbrication then
22        if $\gamma.CRC > r.CRC$ then
23            $r.addr \leftarrow r.addr - \gamma.addr$
24            $f' \leftarrow$ AddFlow $(f, r)$
25        else
26            $f' \leftarrow$ RemoveFlow $(f, \gamma)$
27            $\gamma.addr \leftarrow \gamma.addr - r.addr$
28            $f' \leftarrow$ AddFlow $(f, r)$
29            $f' \leftarrow$ AddFlow $(f, \gamma)$
30    return $f'$
```

6 EVALUATION

The modules described in Section 4 were implemented in JAVA. ODL Lithium was used as the OpenFlow controller and the `L2Switch` project was employed 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 resolved automatically using least privilege and module precedence resolution strategy; and manually using administrator input.

A simple network with topology shown in Figure 9 was implemented on Mininet using a python script. ODL Lithium was used as the OpenFlow controller and the `L2Switch` project was employed to connect to the OVS. OVS and ODL Lithium support both OpenFlow 1.0 and OpenFlow 1.3.

OFAnalyzer was evaluated for correctness by providing it with a number of rules that were known to have conflicts. This rule set contained 100 atomic flow rules, extrapolated from the mininet testbed in Figure 9, with each type of conflict being present. The conflict frequency was as follows: *a*) Shadowing - 10% *b*) Redundancy - 10% *c*) Correlation - 20% *d*) Overlap - 20% *e*) Generalization - 20%; and *f*) Imbrication - 20%. Manual verification showed that our implementation correctly identified flow rule conflicts and classified them, including temporal cross-layer conflicts. While our testing did not detect any false positives, the temporal nature of the imbrication conflict means that the detected conflicts have a fixed validity period. The relationship between the different conflicts were displayed using the visualization techniques discussed. We compared our framework with closest related works such as FortNox [7] and FlowGuard [8], and determined that since 20% of the conflicts were imbrication conflicts, they would not have been detected by earlier works.

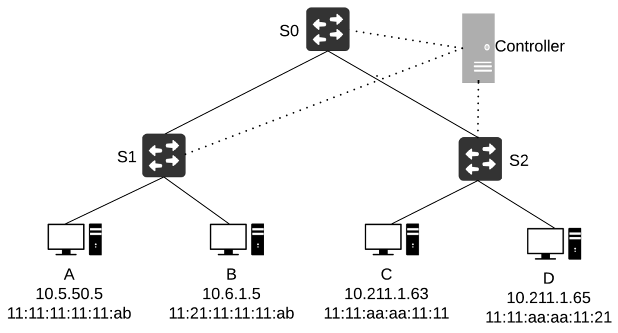


Fig. 9. Verification test topology.

Next, we tested a distributed controller scenario using the application partitioning paradigm. Flows were injected into the controller with weighted priorities giving flows generated from a simulated security application highest preference. The OFAnalyzer extracted flows from the different controllers, and the OFProcessor used the desired weights from the CRC to make decisions as expected. Similar tests were also run using the hierarchical controller paradigm with results matching expectations. Figure 11 shows the running times for the conflict detection algorithm over the same input set of flow rules running on an application partitioning, host partitioning and hierarchical distribution strategies. While all scenarios show a near linear growth in running times, the application partitioning scenario was noticeably faster. We attribute this to the presence of a distributed mesh control plane for the application and host partitioning scenarios while having a hierarchical control plane in the hierarchical controller scenario.

Next, scalability of Brew was tested by performing experiments on a topology derived from the Stanford network [40]. The topology consists of fourteen access layer routers connected using ten distribution switches to two backbone routers. The snapshot of the routing tables and configuration files was translated into equivalent OpenFlow rules resulted in approximately $8,900$ atomic flow rules, which were then used to run scalability tests. Both the conflict detection and resolution algorithms grow in a linear fashion. The time complexity of a lookup on a Patricia trie depends

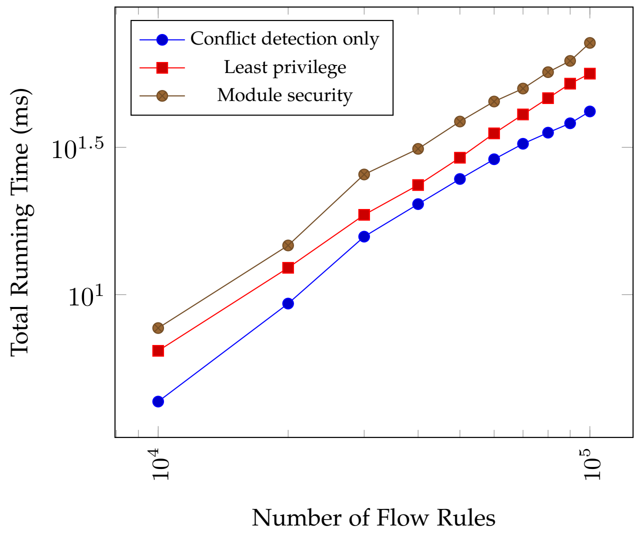


Fig. 10. Brew running time for different conflict resolution strategies.

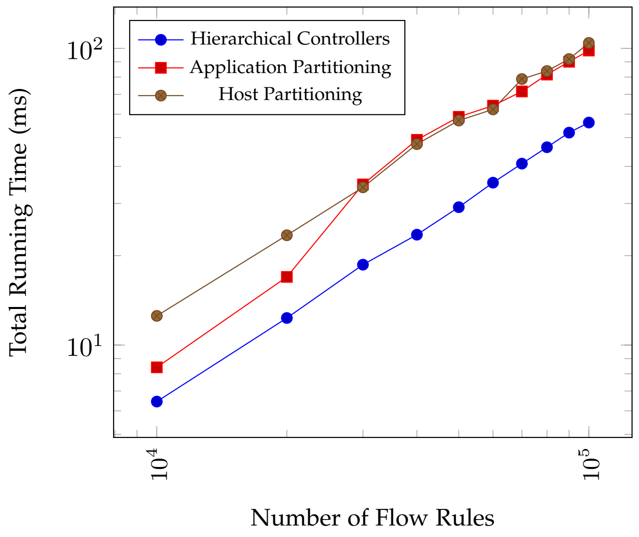


Fig. 11. Brew running time for different decentralization strategies.

on the length of the string (constant in our case) and the number of flow rules; for a worst case runtime of $\mathcal{O}(n)$ [36] and an average case runtime of $\mathcal{O}(\log n)$, 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. 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 8 ms per $1,000$ flow rules). FlowGuard had similar running time of approximately 8 ms per $1,000$ rules. Rules were further replicated and inserted into the system to observe growth of computation time. Figure 10 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 Brew effectively identifies flow rule conflicts and takes corrective action in spite of the large data sets. The results also clearly show worst case of $\mathcal{O}(n)$ running time.

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 9 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 12 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 were resolved using least privilege resolution strategy. As expected, when interpretative conflicts were to be resolved, the transfer took longer, due to additional computational needs on the system. 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

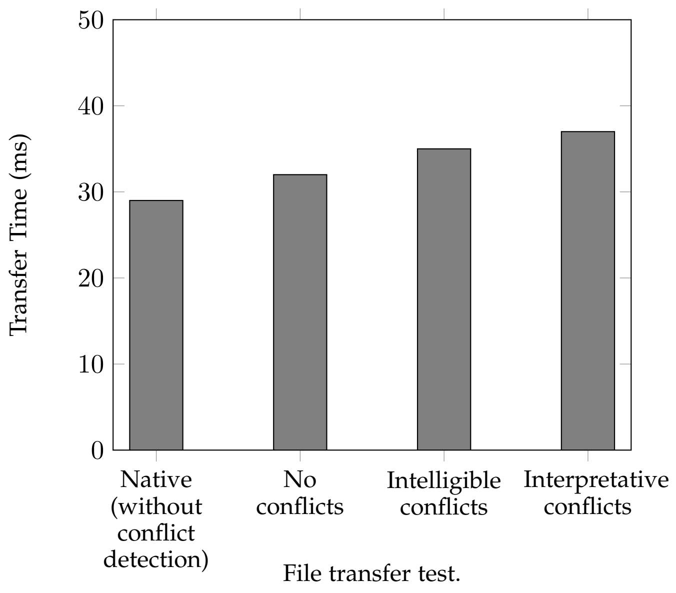


Fig. 12. Network performance impact.

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-based cloud environment since having a conflict free flow table will not only ensure greater confidence in security, but also more optimal packet forwarding processing times.

7 Conclusion & Future Work

Recent advances in SDN creates a unique opportunity to enable complex scientific applications to run on dynamic and tailored infrastructure that includes compute, storage and network resources. This approach provides the performance advantages of strong infrastructure support with little management and deployment costs. However, with several threat vectors for SDN-based cloud environments already being identified, and new threats being developed/discovered every day, comprehensive security implementation in an SDN-based cloud is an issue that needs to be dealt with actively and urgently. Traditional approaches to addressing security issues in such dynamic and distributed environments tried to implement security on individual components, and did not considering security holistically. In a multi-tenant SDN-based cloud environment, presence of various such security applications and network nodes interacting with each other makes it extremely difficult to manage policies and track policy conflicts.

In this work, we augment security enforcement in an SDN-based cloud environment by introducing a framework that monitors and maintains a conflict free environment, including strategies to automatically resolve conflicts. We first present a formalism for the classification of the different types of conflicts in SDN. We address indirect conflicts in flow rules, and present techniques to resolve conflicts automatically in a distributed SDN-based cloud. Several approaches that can be used to resolve flow rule conflicts were presented and their benefits and deficiencies were analyzed. The run time complexity for the framework is linear, and hence scalable to large SDN-based clouds. A novel visualization scheme that assists administrators with decision making has been implemented.

As with any prototype, Brew is not without limitations. In its current avatar, it is unsuitable in a highly dynamic environment because the conflict resolution model that considers temporal nature of the mapping between different address layers in a dynamic SDN cloud is abecedarian. Currently, the topology of the environment is not considered while detecting conflicts. Adding the topology as an input might be able to make the conflict detection more thorough. Finally, the processing time is linear in nature, and optimizing data structures might be able to reduce it further.

Central to the effort to use policy to implement security is to avoid conflicts. There are two potential ways to handle this issue: *a*) eliminate setting up conflicting policies; or *b*) make runtime decisions when a conflict is detected. Since an implementation that has no conflicts would be ideal, using formal verification methods to set up a conflict-free policy is highly desirable. Currently, Brew accomplishes run time decision making for conflicts. Our goal is to use formal language methodologies to ensure conflicting flow rules do not get generated. As part of our next steps, we plan to study using multiple analyzers to share the work load to parallelize processing. Including role-based and attribute-based policy conflicts is a natural extension of this work. Further, we are considering 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 considering flavors of the Brew framework tailored for host based SDN firewalls and a mobile (lightweight) solution for tactical clouds. Future visualization work includes upgrades to provide newer features to assist in scalability. A zoom-in/zoom-out feature aiding in the visualization process and graphs depicting the statistical data gathered from the switches using the OpenFlow protocol could be added. And finally, we plan on expanding the framework to work in an environment with diverse controllers so as to enhance adoption of the framework.

Acknowledgments

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] "OpenFlow V1. 3.1," Tech. Rep., 2013.
[2] S. H. Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.
[3] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.
[4] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and others, "Onix: A Distributed Control Platform for Large-Scale Production Networks." in *OSDI*, vol. 10, 2010, pp. 1–6.
[5] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, 2010, pp. 3–3.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] S. Pisharody, A. Chowdhary, and D. Huang, "Security Policy Checking in Distributed SDN based Clouds," in *2016 IEEE Conference on Communications and Network Security (CNS) (IEEE CNS 2016)*, Philadelphia, USA, Oct. 2016.
[11] 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.
[12] 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.
[13] 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.
[14] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, and others, "Composing Software-Defined Networks," in *NSDI*, 2013, pp. 1–13.
[15] 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.
[16] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed SDN Controllers in a Multi-Domain Environment," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–2.
[17] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and others, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[18] A. Mayer, A. Wool, and E. Ziskind, "Fang: A Firewall Analysis Engine," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000, pp. 177–187.
[19] T. Tran, E. S. Al-Shaer, and R. Boutaba, "PolicyVis: Firewall Security Policy Visualization and Inspection," in *LISA*, vol. 7, 2007, pp. 1–16.
[20] H. Hu, G.-J. Ahn, and K. Kulkarni, "Fame: A Firewall Anomaly Management Environment," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. ACM, 2010, pp. 17–26.
[21] F. Mansmann, T. Gbel, and W. Cheswick, "Visual Analysis of Complex Firewall Configurations," in *Proceedings of the ninth international symposium on visualization for cyber security*. ACM, 2012, pp. 1–8.
[22] 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.
[23] A. Chowdhary, S. Pisharody, and D. Huang, "SDN Based Scalable MTD Solution in Cloud Network," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*. ACM, 2016, pp. 27–36.
[24] J. Liu, Y. Li, H. Wang, D. Jin, L. Su, L. Zeng, and T. Vasilakos, "Leveraging Software-Defined Networking for Security Policy Enforcement," *Information Sciences*, vol. 327, pp. 288–299, 2016.
[25] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.
[26] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee, "No More Middlebox: Integrate Processing into Network," in *ACM SIGCOMM Computer Communication Review*, vol. 40. ACM, 2010, pp. 459–460.
[27] I. Alsmadi and D. Xu, "Security of Software Defined Networks: A Survey," *computers & security*, vol. 53, pp. 79–108, 2015.
[28] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 27–38.
[29] X. Liu, H. Xue, X. Feng, and Y. Dai, "Design of the Multi-Level Security Network Switch System Which Restricts Covert Channel," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*. IEEE, 2011, pp. 233–237.
[30] J. Franois, L. Dolberg, O. Festor, and T. Engel, "Network Security Through Software Defined Networking: A Survey," in *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*. ACM, 2014, p. 6.
[31] 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.
[32] E. Lupu and M. Sloman, "Conflict Analysis for Management Policies," in *Integrated Network Management V*. Springer, 1997, pp. 430–443.
[33] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A Toolkit for Firewall Modeling and Analysis," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.
[34] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter," in *HotNets*, 2009.
[35] J. G. Alfaro, N. Boulahia-Cuppens, and F. Cuppens, "Complete Analysis of Configuration Rules to Guarantee Reliable Network Security Policies," *International Journal of Information Security*, vol. 7, no. 2, pp. 103–122, 2008.
[36] 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.
[37] 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.
[38] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741–748, 2006.
[39] E. M. Reingold and J. S. Tilford, "Tidier Drawings of Trees," *Software Engineering, IEEE Transactions on*, no. 2, pp. 223–228, 1981.
[40] 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.

Sandeep Pisharody is a Ph.D. candidate in Computer Science (Information Assurance) at Arizona State University, Tempe, AZ, USA under the guidance of Dr. Dijiang Huang. Previously, he earned his B.S. Electrical Engineering (distinction) and B.S. Computer Engineering (distinction) from the University of Nebraska, Lincoln, NE, USA in 2004; and MS Electrical Engineering from the University of Nebraska in 2006. Sandeep has over 8 years experience in designing, building and maintaining enterprise and carrier class networks while working in various capacities for Sprint, Iveda, University of Phoenix, Insight and MIT Lincoln Laboratory. His current research interests lie in the areas of secure cloud computing and SDN.

Janakarajan Natarajan received his M.S. degree form Arizona State University, Arizona, USA in 2016 and his B.E. in Computer Science from Anna university (Velammal Engineering College), Chennai, India. His research interests include SDN, the Linux kernel and network security.

Ankur Chowdhary is a Ph.D. Student at Arizona State University, Tempe, AZ, USA. He received B.Tech in Information Technology from GGSIPU, Delhi, India in 2011 and M.S. in Computer Science from Arizona State University, Tempe, AZ, USA in 2015. He has worked as Information Security Researcher for Blackberry Ltd., RSG and Application Developer for CSC Pvt. Ltd. His research interests include SDN, Web Security, Network Security and Machine Learning in field of security.

Abdullah Alshalan received the B.S. degree (with Hons.) in computer science from King Saud University, Riyadh, Saudi Arabia, and the MS degree in computer science from Indiana University, Bloomington, IN, USA, in 2003 and 2009 respectively. While on leave from the College of Computer and Information Sciences, King Saud University, he is pursuing the Ph.D. degree in computer science at Arizona State University, Tempe, AZ, USA. He has 9 years of combined work experience in information security engineering, programming, web development, and teaching. His research interests include computer networks, mobility, information security, and cloud computing.

Dijiang Huang received the B.S. degree from Beijing University of Posts and Telecommunications, Beijing, China, and the M.S. and Ph.D. degrees from the University of Missouri - Kansas City, Kansas City, MO, USA, 1995, 2001, and 2004, respectively. He is an Associate Professor with the School of Computing Informatics and Decision System Engineering, Arizona State University, Tempe, AZ, USA. His research interests include computer networking, security, and privacy. He is an Associate Editor of the Journal of Network and System Management (JNSM) and an Editor of the IEEE Communications Surveys And Tutorials. He has served as an organizer for many international conferences and workshops. His research was supported by the NSF, ONR, ARO, NATO, and Consortium of Embedded System (CES). He was the recipient of the ONR Young Investigator Program (YIP) Award.