Anomaly-Free Policy Composition in Software-Defined Networks

Mohsen Rezvani, Aleksandar Ignjatovic, Maurice Pagnucco and Sanjay Jha
School of Computer Science and Engineering, UNSW Australia
{m.rezvani,a.ignjatovic,m.pagnucco,sanjay.jha}@unsw.edu.au

***Abstract*—Software Defined Networking (SDN) provides considerable simplification of design and deployment of various network applications for large networks. Each application has its own view of network policy and sends its policy to a network hypervisor in which a composed policy is generated from the application policies and deployed into the data plane. A significant challenge for the hypervisor is to detect and resolve both intra and inter policy anomalies during the policy composition. However, current SDN compilers do not consider the policy anomalies well and generate large number of unnecessary rules for the data plane. This leads to a considerable inefficiency in both policy composition and policy deployment. In this paper, we propose a novel framework for policy composition in a SDN hypervisor which takes into account both inter and intra policy anomalies. Moreover, we augment the framework with an efficient insertion transformation mechanism which allows the applications to issue rule insertion and priority change updates. Our evaluation shows that our method is several orders of magnitude more efficient than the state of the art in both policy composition and compiling the rule insertion updates.**

I. Introduction

Software Defined Networking (SDN) is transforming traditional network architectures to more flexible and programmable platforms by decoupling the control logic from the forwarding (data) layer. A logical SDN architecture includes three distinct layers: application, control and forwarding (data) [1]. Network applications at the top of this multi-layer architecture can define network policies based on a global view of the network provided by software-based controllers in the control layer. The controllers enforce network policies in the data layer by translating application defined policies into low-level and identifiable rules in network devices. The OpenFlow protocol [2] is one of the earlier and more popular communication standards between the control and data layers.

The SDN multi-layer architecture allows multiple applications or even multiple administrators to specify the network policy simultaneously. Each application can take advantage of the global network view to effectively define its network policy as a sequence of OpenFlow rules. The controller then, as a *hypervisor* integrates the policies received from different applications based on a policy composition strategy. The strategy specifies how to use three common binary operators: *parallel* (two policies can be applied at the same time), *sequential* (the second policy is processed after the first one) and *override* (the second policy is applied only on the traffic which is not matched by the first policy) to combine the policies [3]. As a result of such policy composition, the controller generates a set of prioritized OpenFlow rules, called *composed policy* and installs such a policy into the data plane (as shown in Fig. 1).

ISBN 978-3-901882-83-8 © 2016 IFIP

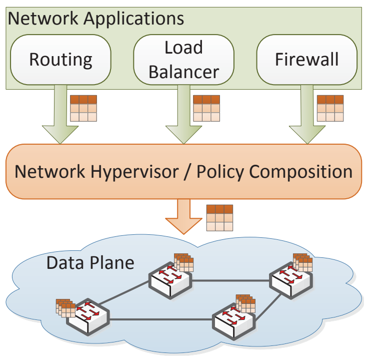


Fig. 1: Policy composition in a SDN hypervisor.

A naive policy composition method is to recompute and reinstall the composed policy every time an application updates its policy. Jin et al. [4] recently proposed *CoVisor* which incrementally updates the policy without shifting existing rules. CoVisor limits the policy updates to only two operations: *add* a new rule and *delete* an existing rule. However, in many applications, such as firewalls, an administrator can *insert* a new rule in the middle of existing rules. Such insert rule operation can lead to a shift of many rules if there is no empty space for the priority of the new rule [5]. Accordingly, applying a rule insertion in CoVisor needs in average $\frac{n}{2}$ delete and $\frac{n}{2}$ add operations in the base policy, where n is the number of rules in the policy. Note that a composition operator, such as parallel operator, after such many updates in the application policy may generate $O(n^2)$ updates in the composed policy.

Another important challenge in SDN policy management is to detect and resolve policy anomalies, such as redundancies and conflicts, in the application policies and then combine them to make an anomaly-free composed policy. This not only reduces the number of rules in the application-level policies [6], but also considerably improves the efficiency of the policy composition as it prevents cascading the anomalous rules into the composed policy. Although the anomaly detection within an individual application policy (intra-policy anomalies) has been well investigated in the literature [6], [7], [8], [9], the SDN multi-layer architecture makes the anomaly detection more difficult. This is because existing anomalies among policies (inter-policy anomalies) must be considered. Our experiments show that the policy anomaly detection is more significant in a SDN hypervisor as its policy composition can quadratically propagate the existing intra-anomalies into the composed policy. This can result in deploying many unnecessary rules into the data plane which leads to not only inconsistency in the network policy but also significant inefficiency due to deployment of these rules.

To address the above challenges, we first propose a novel anomaly-free policy model which help us to efficiently detect and resolve anomalies for policy updates. We maintain a separate model for each application-level policy. Then, we define the policy composition operators over our policy model which helps us to generate a model for the composed policy. Thus, the results of policy composition is a policy model which is also an anomaly-free model. We then propose a straightforward algorithm to translate the model to low-level OpenFlow rules. Moreover, we leverage our policy model to efficiently translate the rule insertion updates received from the application policies. The policy model efficiently emits the sequence of prioritized rules which reduces the complexity of both anomaly detection and policy composition.

In summary, we make the following contributions.

- We propose a new model for OpenFlow-based policies which allows to efficiently obtain the dependencies between OpenFlow rules;
- We develop a formal mechanism to detect and resolve both intra and inter-application policy anomalies which leverages our OpenFlow policy model to reduce the complexity of the detection process;
- We develop a new algorithm to incrementally compose inserting updates which eliminates unnecessary shifting in both individual and composed policies.

We provide a comparative evaluation of the performance of our algorithms with the state of the art in SDN policy composition. The results show that our method significantly improves the efficiency of the policy composition by reducing the update length several orders of magnitude compared to proposed method in [4]. Moreover, the proposed insertion translation considerably increases the performance of the naive approach for handing insertion updates.

The rest of this paper is organized as follows. Section II presents the related work. Section III describes the details of our policy model. Section IV presents our anomaly-free policy composition system. Section V describes our experimental results. Finally, the paper is concluded in Section VI.

II. Related Work

Anomaly detection in traditional access control policies, such as firewalls, has been extensively studied in the research community [10], [6], [7], [8], [11]. Al-Shaer and Hamed present a set of algorithms to discover simple pairwise anomalies in centralized and distributed firewall rules [6]. Inconsistencies and inefficiencies among multiple rules are treated in [7], [9]. Adao et al. [11] propose *Mignis*, a declarative policy language to specify a Linux firewall, Netfilter configurations. The Mignis tool is tightly integrated with Netfilter and is hard to use for the OpenFlow policies.

Several SDN policy languages, such as Frenetic [12], NetKAT [13] and Pyretic [3], have been proposed. They use different batch mechanisms for policy composition which make a large number of updates for each update in an application policy. Wen et al. [5] introduced an incremental policy update for Frenetic policies. The proposed method maintains a dependency graph and scattered priority distribution for each policy. Our method, instead of a using a dependency graph, maintains an anomaly-free model which efficiently handles the priority updates, such as insertion, without maintaining any priority distribution. Recently, Jin et al. [4] proposed CoVisor which employs a simple algebra for priority assignment in an incremental policy composition. However, CoVisor neither considers policy anomalies nor rule insertion updates. While our method uses the algebra proposed in CoVisor, it also takes into account both policy anomalies and insertion updates.

Han et al. [14] proposed a multi-layer policy management framework for SDNs. However, the proposed method for intra-policy anomaly resolution makes the policy enforcement a nondeterministic task which can affect the rules' semantics and change the intention of policy definition [8]. Moreover, the proposed method for inter-policy anomaly detection does not consider the priority assignment in the incremental policy composition. Dwaraki et al. [15] proposed *GitFlow*, a conflict-free flow repository management for SDNs. However, GitFlow considers neither the total conflicts in the application policies nor the conflicts during the policy composition. Shin et al. [16] present FRESCO, a framework for developing and deploying network security applications for OpenFlow networks. However, its conflict detection module only considers simple pairwise rule conflicts in the data plane level. Prakash et al. [17] recently proposed policy graph abstraction (PGA), a high level policy graph abstraction which provides a conflict detection and resolution mechanism. However, PGA is a whitelisting model while the OpenFlow policies in an SDN hypervisor can contain blocking rules. Smolka et al. [18] proposed a fast compiler for NetKAT which introduces forwarding decision diagrams (FDDs) to improve the efficiency of BDDs for encoding the packet headers. Although our experiments show a promising performance with BDDs, in our framework one can also use FDDs for conflict detection and resolution.

III. SDN Policy Model

We assume that each application defines its policy as a set of flows defined in OpenFlow specification [19]. In this paper, the term *rule* refers to a flow in OpenFlow specification.

A. OpenFlow Rule

Without loss of generality, we represent an OpenFlow rule r with three components: 1) a *priority*, denoted as $r.priority$, is a non-negative integer value used for matching precedence of the rule within a policy; 2) *rule match*, denoted as $r.match$ which is a set of *matching fields* for specifying a set of packets; and 3) *actions*, denoted as $r.actions$ which is a set of instructions to apply on the packets. A rule r defines how a set of packets specified in $r.match$ is treated in the network.

In the basic model, the matching fields represent information about source and destination of the matched packets. Thus, we can simply divide these fields into two sets where each of them is called a *peer*. A combination of these fields defines the source (destination) of the matched packets and is called *source (destination) peer*, denoted as $r.speer$ ($r.dpeer$) for a rule r. In other words, source (destination) peer is a generalization to specify the source (destination) sides of the matched packets. Specifically, $r.speer$ ($r.dpeer$) is a combination of matching fields including source (destination) MAC, source (destination) IP, source (destination) port, and protocol. A matching field x for a peer p is denoted as $p[x]$. Note that some fields in OpenFlow, such as protocol and VLAN identifier, are neither assigned to source nor destination as they are common for both peers. Thus, we repeat their values in the specification of both peers if a rule is defined by these fields.

In order to model a peer of a rule, we use the idea of binary representation of packet headers proposed in [9], [20], [21]. To this end, we first encode each matching field as a bit stream and a peer is then represented as a bit stream obtained from a combination of bit streams of its matching fields. In the basic model, the representation of a matching field in a peer contains: a MAC address as a stream of 48 bits, a network address as a stream of 32 bits, a port as a stream of 16 bits, and a protocol as a with of 8 bits. Thus, a peer is encoded as a stream of $48 + 32 + 16 + 8 = 104$ bits.

Using the above encoding, a matching field x in a peer p is represented as a propositional logic formula, denoted as $p[x].formula$. We define n variables to make the formula from a stream of n bits. The formula is conjunction of the variables for every bit set 1, its negation for every bit set zero, and nothing for every bit set *don't care* [9]. Now rule r is represented by a conjunction of the formulas obtained for its matching fields. Thus, $r.match.formula = \bigwedge_{x \in r.match} x.formula$ and the match formula contains at most $104 * 2 = 208$ variables.

A rule match can be defined based on conjunction of matching fields. All fields but network addresses define either all values using a *wildcard* or only one specific value. A network address is represented by a CIDR domain which defines an arbitrary set of IP addresses. Thus, a matching field can be specified by a set of values. Now we show that all the fields hold the DisjointOrSubset property which is employed to propose a hierarchy representation of different matching fields in a policy. Note that OpenFlow proposes a bitmask feature which allows matching single bits of a filed [19]. DisjointOrSubset holds when there is no such bitmask in the fields. Supporting the bitmask is our future work.

Proposition 1 (DisjointOrSubset). *Each two values in a matching field are either disjoint or one is a subset of the other. More formally, assume that p_1 and p_2 are two peers and x is a matching field defined according to our OpenFlow rule model. The following relation holds:*

$$\forall p_1, p_2 : (p_1[x] \cap p_2[x] = \emptyset) \lor (p_1[x] \cap p_2[x] = p_1[x]) \lor (p_1[x] \cap p_2[x] = p_2[x]). \quad (1)$$

Proof. Since all matching fields but network address define either wildcard or only one specific value, it is obvious that they satisfy the property. A network address field is specified as a masked CIDR domain which contains an IP address and a *mask* which is an integer value in the range of [0,31]. Clearly, if two network domains contain non-overlapping IPs, they are disjoint and otherwise one with smaller mask is a subset of the other one. Thus, all the matching field holds the property. □

B. OpenFlow Policy

Our idea for modeling a network policy is the fact that the main source of inefficiency in a SDN hypervisor is the set operations, such as union and intersection of match rules, needed during both policy composition and anomaly detection. We propose a graphical model to represent an OpenFlow policy which is inspired by PGA [17]. Since our solution is deployed within a SDN hypervisor (as shown in Fig. 1), our policy model is required to 1) specify prioritized rules while PGA only supports a set of rules without any priority; 2) specify different actions according to OpenFlow specification [19] which includes conflicting actions, such as *Forward* and *Drop*, while PGA is limited to a whitelisting policy containing only permit rules; and 3) incrementally compose policies and generate minimum updates for deploying into the data plane, while PGA supports batch policy composition.

An OpenFlow policy is a set of rules $P = \{r_1, r_2, \ldots r_n\}$, where $r_i = (r_i.priority, r_i.speer, r_i.dpeer, r_i.actions)$, $(1 \leq i \leq n)$ represents the i^{th} rule in the set. Now we introduce the Policy Semantics Graph (PSG) to model an OpenFlow policy.

Definition 1. ***(Policy Semantics Graph)*** *A PSG is a directed graph, generated from an OpenFlow policy P. The vertices are the peers in the rules in P. Also, there is an edge between two vertices corresponding to peers p_1 and p_2 if and only if $\exists r \in P, r.speer = p_1 \land r.dpeer = p_2$. An edge represents a rule in the policy and consists of the priority and actions of the corresponding rule.*

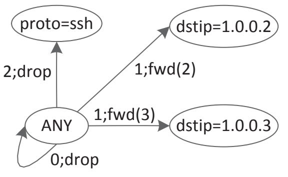
Corollary 1. *A PSG model has no multiple edges between any two vertices (thus a PSG is not a multigraph).*

Proof. Follows directly from the fact that PSG has one and only one edge corresponding to each rule in the policy. □

Corollary 1 shows that PSG is a replica-free policy model. Thus, the model automatically eliminates the fully replicated rules from the policy. Note that a policy may contain other types of rule redundancy, described in Section IV-C. Figures 2(a) and 2(b) show an example of an OpenFlow policy and its corresponding PSG model, respectively.

Priority	Match	Actions
2	proto=ssh	drop
1	dstip=1.0.0.2	fwd(2)
1	dstip=1.0.0.3	fwd(3)
0	*	drop

(a) An OpenFlow policy.



(b) A PSG model.

Fig. 2: An OpenFlow policy and its PSG model.

C. SDH Hypervisor

As shown in Fig. 1, each application maintains a network policy specified as an OpenFlow policy. The hypervisor also maintains a policy obtained from a combination of the application policies, called composed policy. As each application submits its policy updates, such as add/delete/insert rules, the hypervisor accordingly updates the composed policy and installs the final updates into the data plane. Note that the details of translating the composed policy into physical policies, including mapping the virtual network to physical network is out of scope of this paper. The network administrator configures the composition between each two application policies using three operators: *sequential*, *parallel*, and *overriding* [12]. In this section, we briefly review these operators.

a) Parallel Composition: The parallel composition between policies P_1 and P_2 applies both policies over all incoming packets and then computes the union of their output packets. For example, one can combine the policies from *Monitoring* and *Routing* applications using the parallel operator. Accordingly, the actions obtained from both policies are applied on any incoming packet.

b) Sequential Composition: The sequential composition between policies P_1 and P_2 applies P_2 after P_1 over the incoming traffic. To this end, the hypervisor first applies policy P_1 on the traffic and it then applies policy P_2 on the resulted traffic. For example, and administrator can use the sequential composition to combine the policies from *Load Balancer* and *Router* applications. The incoming packets first pass the load balancer policy which might manipulates the packet header. The routing policy is then applied on the new packet headers.

c) Overriding Composition: Such composition of policies P_1 and P_2 applies policy P_2 over the incoming traffic which does not match P_1. The overriding operator can be used to define a default policy.

IV. Anomaly-Free Policy Composition

A. Solution Overview

The main idea behind our policy composition is to model the application policies using PSG and then combine the PSG models to obtain a new PSG which specifies the composed policy. We assume that each application sends a list of OpenFlow rules as *policy updates* to our system. The updates may request to add, delete, or insert rules into the application policy. Our system compiles the updates and generates a new list of rules for the data plane.

Fig. 3 shows our anomaly-free frameworkcontaining two layers: in the upper layer there is a policy manager for each application and in the lower layer there is one policy manager for the composed policy. Each policy manager maintains a PSG to specify its policy. The updates from an application first enter into the corresponding policy manager in which the updates are checked for intra-policy anomalies such as rule redundancy. The updates passed from the anomaly detection are considered for updating the PSG model in the manager which generates updates for the lower layer. The composed policy manager employs the application-layer PSG models to apply a typical policy composition and obtain a list of add/delete rules to update the composed policy. After that, the updates are checked for both intra and inter policy anomalies, the composed PSG is updated accordingly, and finally the updates are sent to the data plane.

It is worth noting that we augmented the application policies to submit updates including rule insertion. This provides an opportunity for an application manager to either insert rules in the middle of the policy or update the priority of rules. This is a common requirement supported in traditional rule-based security tools such as firewalls and VPNs. As one can see in Fig. 3, an application can generate insertion updates and the application policy manager efficiently transforms such updates into several Add/Delete updates. Thus, the composed policy manager only accepts Add/Delete updates. The detailed insertion transformation is described in Section IV-E.

B. Policy Construction

As described, each policy in our solution is represented by a PSG. Thus, the hypervisor maintains one PSG for every application and one for the composed policy. It is clear that the main operations with high time complexity in both policy composition and anomaly detection are set theoretic operations, such as union and intersection, among rule matches. Thus, we employ a multidimensional Patricia trie to maintain the list of peers used in each policy. The multidimensional trie

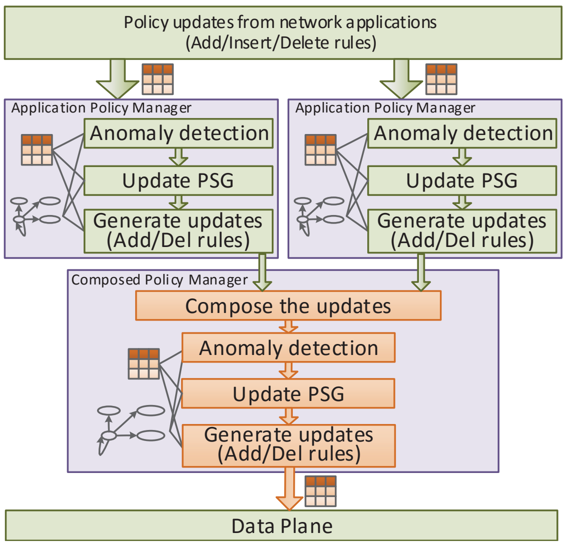


Fig. 3: Our policy composition framework.

is widely used for packet classification [22] and SDNs [23]. We instead use a Patricia trie which is a compressed trie and stores data in every node. Since all the matching fields hold the DisjointOrSubset property (as shown in Proposition 1), we can efficiently store them in the trie. In our data structure, each level in the trie is corresponding to a matching field. Also, each node which contains data, has a link to the root node of a lower-level trie specifying the next matching field. This data structure helps us to obtain the union and intersection using postfix and prefix functions in the tries. Note that each field is represented by a bit stream which help us to make a trie for all values of the field within a policy.

In order to build the PSG model, we employ an incremental method in which for any new rule, the corresponding nodes and edge are created in the model. We also use an adjacency list to represent the PSG model while its nodes are stored in a multidimensional trie.

As described in the previous section, in order to implement the sequential composition, we need the intersection between matching packets after applying the actions of rules. In order to improve the computational complexity of the sequential composition, we maintain another trie which specifies the results of applying all the rules within the policy. Using this idea, we sacrifice the space complexity to reduce the time complexity of the sequential composition to same as the complexity of the parallel composition.

C. Anomaly Detection

Al-Shaer et al. [6] introduced four types of pairwise anomalies among rules in a policy: *Shadowing*, *Correlation*, *Generalization* and *Redundancy*. Basi et al. [24] also classified the anomalies into two categories: *conflict* where a packet is matched with multiple rules with conflicting actions, and *suboptimality* where there is a rule such that its removal has no effect on the policy. We extend these pairwise anomalies to hidden anomalies between a rule and a set of other rules in the policy, called *total anomalies* [9]. Now, we leverage the PSG model to efficiently detect the anomalies in a policy.

We concentrate on redundancy anomalies as the detection algorithms for other anomalies are mostly similar. Moreover, this avoids adding the redundant rules into the policy which reduces the number updates for the policy. Note that avoiding the redundant rules in the application policies can quadratically decrease both the number of rules in the composed policy and the number of updates installed in the data plane.

Definition 2. ***(Simple Redundancy)*** *An OpenFlow rule is simply redundant in a policy if its rule match is overlapped with an existing rule with a higher priority in the policy. More formally, rule r is simply redundant in policy P if and only if*

$$\exists r_i \in P : r.priority \leq r_i.priority \wedge r.speer \subseteq r_i.speer \wedge r.dpeer \subseteq r_i.dpeer.$$

A naive algorithm for detecting simple redundancy of a new rule is to traverse over all rules with a higher priority than such rule. Thus its complexity would be in $O(n)$, where n in the number of rules in the policy. However, one needs to investigate only rules whose both peers are supersets of the peers of the new rule. We leverage the trie structure to obtains such rules for checking the simple redundancy. Clearly, if the new rule is redundant, it is ignored and the system generates a message to notify the administrator of this redundancy.

Definition 3. ***(Total Redundancy)*** *An OpenFlow rule is totally redundant in a policy if its rule match is overlapped with a set of existing rules with higher priority in the policy. More formally, rule r is totally redundant in policy P if and only if the following formula is a tautology.*

$$r.match.formula \rightarrow \bigvee_{\substack{r_i \in P \\ r.priority \leq r_i.priority}} r_i.match.formula \qquad (2)$$

Similarly to the detection algorithm for the simple redundancy, one can search over all rules with a higher priority than the new rule to construct the propositional formula on the right hand side of the implication in Eq. (2). This equation then is transformed into a propositional logic formula. If the formula obtained from the equation is a *tautology* (a proposition that is always true), the new rule is totally redundant. Now the redundancy detection is transformed to a theorem proving problem in propositional logic. A typical method for theorem proving in propositional logic is using Binary Decision Diagrams (BDDs) [25].

As we have shown, an OpenFlow rule can be encoded as a propositional logic formula using maximum 208 variables. This formula can be represented as a 208-variable BDD. Accordingly, we can build a BDD to represent Eq. 2 for each new rule. After that, the obtained BDD is checked to validate the equation. Although the procedure seems straightforward, the size of the BDD exponentially increases as the number of rules in the policy increases [25]. This is because of many variables used in the decoding of an OpenFlow rule match.

The idea for reducing the size of the BDD obtained from Eq. (2) is to consider only effective rules for transforming the equation to a propositional logic formula. Assume that the hypervisor wants to add a new rule r to policy P. An existing rule $r_i \in P$ is effective for the redundancy detection of rule r if all of the following conditions are held:

- The priority of r_i is greater than or equal to the priority of r, $r_i.priority \geq r.priority$;
- There exists some packets matched by both rules. In other words, the formula $r.match.formula \wedge r_i.match.formula$ is not a contradiction (a proposition that is always false);
- Assume the current formula obtained for the right side of Eq. (2), is denoted by f. Adding rule r_i to f is effective if formula $r_i.match.formula \rightarrow f$ is not a tautology.

It is clear that if an existing rule r_i satisfies the above conditions, it certainly contributes to the right side of Eq. (2). The last condition avoids adding rules which already overlapped with the obtained formula for the right side. By combining the last two conditions, we only check if formula $(r.match.formula \wedge r_i.match.formula) \rightarrow f$ is a tautology. Algorithm 1 shows the procedure for detecting total redundancy of a new rule r with policy P.

Algorithm 1 Total redundancy detection.

```
1: procedure TotalRedundancy(PSG $P$, Rule $r$)
2:     $f \leftarrow BDD(False)$
3:     for all rule $r_i \in P$ do
4:         if $r.priority \leq r_i.priority$ then
5:             $f' \leftarrow BDD((r.match \wedge r_i.match) \rightarrow f)$
6:             if $f'$ is not a tautology then
7:                 $f \leftarrow BDD(f \vee f')$
8:             end if
9:         end if
10:    end for
11:    $f \leftarrow BDD(r.match \rightarrow f)$
12:    if $f$ is a tautology then
13:        return $True$
14:    end if
15:    return $False$
16: end procedure
```

D. Policy Composition

Jin et al. [4] proposed CoVisor, an incremental policy composition algorithm for making small changes to the composed policy every time an application policy changes. Basically, an incremental composition does not need to recompute and reinstall all rules in the composed policy for every single update in an application policy. Instead, it leverages a priority assignment mechanism to generate and install only rules which are related to the update. We employ the calculus proposed in CoVisor for prioritizing the new rules in the composed policy and show how our PSG model improves the efficiency of the policy composition. Note that the overriding composition is implemented using a straightforward algorithm as there is no need to apply any set operators among rule matches. Thus, we take advantage of our PSG model to implement the parallel and sequential compositions.

a) Parallel Composition: In order to combine two policies P_1 and P_2 using the parallel composition, for any two rules $r_1 \in P_1$ and $r_2 \in P_2$ we add a new rule r_k along two original rules r_1 and r_2 to the composed policy. The rule r_k is added into the composed policy if the rule matches of two base rules are not disjoint and r_k is thus obtained as

$$r_k.match = r_1.match \cap r_2.match$$
$$r_k.actions = r_1.actions \cup r_2.actions$$
$$r_k.priority = r_1.priority + r_2.priority$$

Without loss of generality, we assume that a request arrives to add a new rule r_1 to application policy P_1 and the hypervisor wants to obtain all corresponding updates in the composed policy. A naive algorithm is to traverse all rules in P_2 and create all possible rules, such as r_k described above. However, we leverage the trie structure of peers in the PSG model of P_2 to efficiently obtained all rules in P_2 which has intersection with r_1. To this end, we first obtain two sets SP and DP which are overlapping peers with $r_1.speer$ and $r_1.dpeer$, respectively. Then, the edges in PSG from a node in SP to a node in DP represent all the rules intersected with r_1. Note that the updates generated from the policy composition are used to update the PSG model of the composed policy. Clearly, some of the updates may be removed in the anomaly resolution of the PSG model.

b) Sequential Composition: Similar to the parallel composition, we assume that new rule r_1 is added to policy P_1 and the hypervisor can use a similar method to obtain the updates for sequential composition of P_1 and P_2. The only difference is that instead of $r_1.speer$ and $r_1.dpeer$ the hypervisor uses the peers $r_1.speer'$ and $r_1.dpeer'$ which respectively correspond to $r_1.speer$ and $r_1.dpeer$ after applying the actions of r_1.

In the case that a new rule r_2 is added to policy P_2, the hypervisor must obtain the rules in P_1 which their matches after applying their actions have intersection with $r_2.match$. As discussed in the model construction phase, the hypervisor maintains two PSG models for each policy, one for the original policy and another for modeling the peers after applying the actions. We employ the second PSG model of P_1 to obtain a subset of rules in P_1 must be sequentially combined with r_2.

E. Insertion Transformation

The methods described in the previous sections implement both add and delete rules to/from an application policy and consequently to/from the composed policy in the hypervisor. Since the flow tables in an OpenFlow switch support redundant priority for the flow entries [19], adding a rule is implemented without changing the priority of other rules in the policy. However, many rule-based security systems allow the administrator to manipulate the priority of existing rules as well as inserting a rule in a specific position with the policy.

For example, assume that the administrator of the policy shown in Fig. 2 wants to forward all the packets coming from source IP 2.0.0.5 to port number 5 except for the *http* packets which must be dropped. Now, the administrator can simply insert a rule "`2;srcip=2.0.0.5;fwd(5)`" between the first two rules in the policy. To this end, the first rule must be shifted in order to make a free space for the new rule. This is because there is a rule with an identical priority and with an overlapped rule match with the new rule in the policy.

It is worth noting that the inserting operation is a very beneficial feature in rule-based applications as the administrator can locally decide about the position of a new rule without checking whole of the rules in the policy. A naive method to implement the insert operation is to first shift all the higher priority rules in order to make free space for the new rule. After that, the insert operation is transformed into an add operation. Accordingly, if there are n rules with a higher priority than the new rule, this method first removes these n rules, adds them while increments their priority, and finally add the new rule in the free space. Thus, the naive method needs $2n + 1 = O(n)$ update operations. Note that such number of updates generated for updating an application layer policy can lead to an update with a quadratic length, $O(n^2)$ for either a parallel or sequentially composed policy.

It is clear that the main objective of the rule priorities is to prioritize the overlapping rules in the match process as the OpenFlow policy uses a *first match* mechanism. In other words, such priority can be disregarded when the match rules are disjoint. By leveraging this fact we formulate our approach based on two propositions: 1) we can limit the propagation of any priority update to only the rules intersected with the updated rule; and 2) if the policy contains a free space for the inserted rule, we need no rule shifting. Note that we augment the application policy updates allows an administrator to insert rules. The hypervisor uses our approach to transform the insert operation into a list of add/delete updates for such policy. It then sends these updates to the policy composition module to generate corresponding updates for the composed policy.

Algorithm 2 shows our approach for transforming an insert update requested for an application policy into a list of add/delete updates. The method proposed for model construction is used for applying the add/delete updates in the application policy. Moreover, the composed policy is updated for the updates based on the composition algorithms described in the previous section.

Algorithm 2 Transforming an insert rule into add/delete rules.

```
Input: $P$: a PSG, $r$: a new rule to be inserted into $P$
Output: $U$: A set of add/delete updates
 1: procedure TRANSFORMINSERT(PSG $P$, Rule $r$)
 2:     $U \leftarrow \{Add(r)\}$
 3:     if $\nexists r' \in P, r'.priority = r.priority$ then
 4:         return $U$
 5:     end if
 6:     for all rule $r' \in P$ do
 7:         if $r.priority \leq r'.priority$ then
 8:             $f \leftarrow BDD(r.match.formula \wedge r'.match.formula)$
 9:             if $f$ is not a contradiction then
10:                 $U \leftarrow U \cup \{Del(r')\}$
11:                 $r'' \leftarrow Clone(r')$
12:                 $r''.priority \leftarrow r''.priority + 1$
13:                 $U \leftarrow U \cup \{Add(r'')\}$
14:             end if
15:         end if
16:     end for
17:     return $U$
18: end procedure
```

F. PSG to Flow Table

Although we propose an incremental method for updating the flow tables in the data plane, a hypervisor might regenerate and reinstall the rules into the data plane. Thus, we need to efficiently translate the PSG model into forwarding tables that represent packet processing in the switches.

As we described, a path from root to a leaf node in the multi-dimensional trie represents an existing peer in the policy. Such a leaf node has an edge to another leaf node for each rule in which the rule participates as a source peer. Thus, in order to generate a set of forwarding rules from a PSG model, one can use a DFS-like algorithm for traversing the whole of the trie and generate the rules for each source peer. Note that the PSG model contains the priority of the rules and there is no need to sort the rules before deploying them into the data plane. Moreover, the forwarding rules generated using this approach from a PSG model is anomaly-free as we resolves the anomalies during the model updating process.

V. Experimental Evaluation

A. Experimental Environment

We implemented a prototype of our method on the top of CoVisor[1] which itself is a part of the OpenVirteX network hypervisor [26]. We selected this platform for two reasons. First, this allows us to conduct a comparative evaluation against CoVisor, a recently proposed policy composition method [4]. Second, our policy composition works independently and is compatible with the built-in virtualizations in both OpenVirteX and CoVisor. We used the `PatriciaTrie` API[2] to implement the trie for each matching field. We extended the library to also give us a sorted map of objects that are not disjoint by a key. This helps to efficiently detect both simple and total anomalies. We also used the `JavaBDD` library[3] to manipulate BDDs needed for our anomaly detection mechanism.

We assume that the applications' administrators generate their policy update requests as a list of add/insert/delete rules. Thus, in all experiments we evaluate the efficiency of the hypervisor for generating and installing the corresponding updates for the composed policy based on four types of metrics: 1) **update length** which is the average number of updated flows in the composed policy per each update in the application policy; 2) **total length** which is the composed policy length after compiling the updates in the application policies; 3) **compile time** which is the average elapsed time needed to compile one update from an application policy into the composed policy; and 4) **total time** which is the total elapsed time needed to compile one update and install the corresponding generated updates into the data plane.

We conducted experiments to evaluate the performance of the proposed policy compositions and insertion transformation. In all these experiments, the application policies are obtained based on the filter sets generated by ClassBench [27] and we randomly create the rule updates. All the experiments were conducted on an iMac PC with 2.00GHz Intel Core 2 Duo processor and 4GB RAM running Ubuntu 12.04 LTS. The program code was written in Java with JDK 1.7.

B. Policy Composition

In order to conduct a fair comparative evaluation of the policy composition, we follow the evaluation method proposed in [4]. Accordingly, in this section we consider a network with two applications where the hypervisor composed their policy using either a parallel, sequential or overriding composition operator. To measure the efficiency metrics, we first install the base policy of both applications into the hypervisor, and then measure the performance of policy updating by adding 10 uniformly randomly selected rules into both applications. The process is repeated 100 times and then results are averaged. We also vary the number of rules in the base policies for both application to investigate the performance of the composition algorithms in different policy lengths. For all experiments, we measure the performance metrics for both the proposed method, called *PSG* and *CoVisor* proposed in [4].

Fig. 4 shows the performance results of both approaches for a parallel composition between two applications such as Monitor and Router. As one can see in Fig. 4(a), the rule update overhead in CoVisor linearly increases as the number of rules increases in the base policies while the overhead is approximately steady for PSG. Moreover, the number of flows generated per each update in PSG is always less than CoVisor. This can be explained by the fact that the number of anomalous rules increases in the larger base policies and consequently this dramatically raises the number of unnecessary rules generated in the composed policy. A similar trend can be observed in the composed policy length shown in Fig. 4(b). As we expected, the total number of flows in the composed policy generated by PSG is considerably less than the policy generated by CoVisor. This is because of many unnecessary flows in both base and composed policies were not treated in CoVisor.

Figures 4(c) and 4(d) show the compile time and total time of composition per rule update, respectively. The results show that the policy compilation using PSG takes longer than CoVisor. This is due to additional processing for anomaly detection and resolution based on algorithms described in Section IV-C. However, PSG is significantly faster than CoVisor according to the total time of policy composition shown in Fig. 4(d). This can be explained by the fact that PSG sacrifices a few milliseconds to resolve the anomalies and consequently it considerably reduces the number of updates needed to be installed in the data plane. Since the hypervisor needs to communicate with the switches to install the updates, reducing the update length through eliminating the unnecessary rules leads to a significant efficiency in the network.

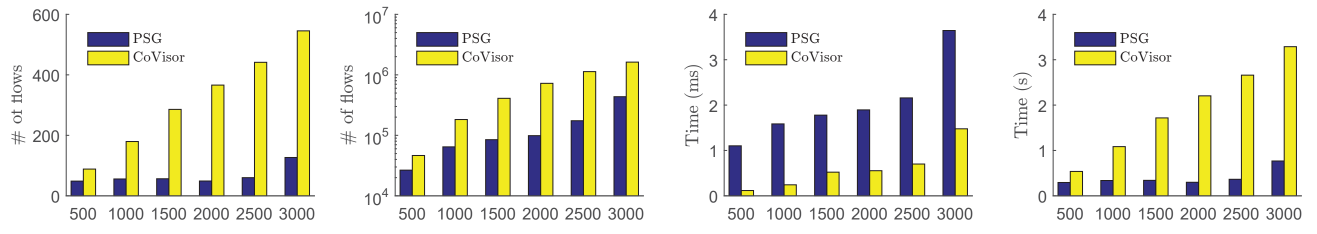
In the second scenario, we assume a sequential composition between two applications such as a Firewall and a Router. Fig. 5 shows the performance results of this composition. Again, PSG is several order of magnitude efficient than CoVisor in terms of number of rules generated per each update as well as the length of composed policy. Although that PSG is slightly slower than CoVisor in compilation of a sequential composition, it is significantly faster than CoVisor according to the total composition time as it considerably filters the anomalous rules.

Fig. 6 illustrates the performance results of an overriding composition between two application policies. As one can see in Fig. 6(a), CoVisor always generates one update rule for the composed policy while the average number of update rules generated by PSG is less than one. Note that the maximum updates per one rule added in overriding composition is one rule in the composed policy as we have employed an incremental mechanism. Thus, since PSG is an anomaly-free composition method, it eliminates the anomalous rules and consequently its average update length is less than one.

[1] http://covisor.cs.princeton.edu/

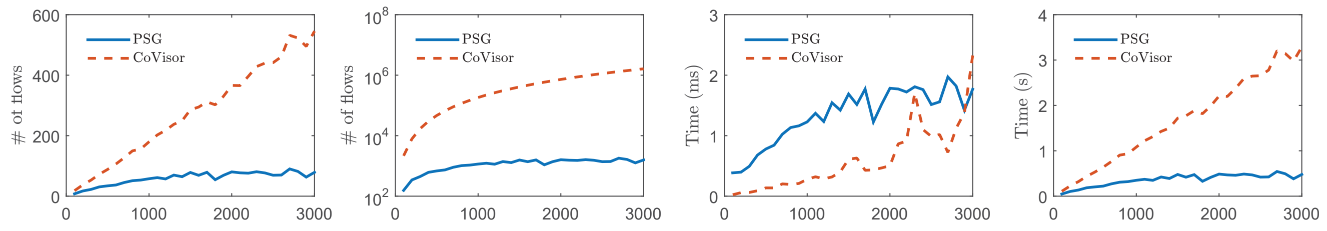
[2] http://docs.oracle.com/javase/6/docs/technotes/guides/collections/

[3] http://javabdd.sourceforge.net/



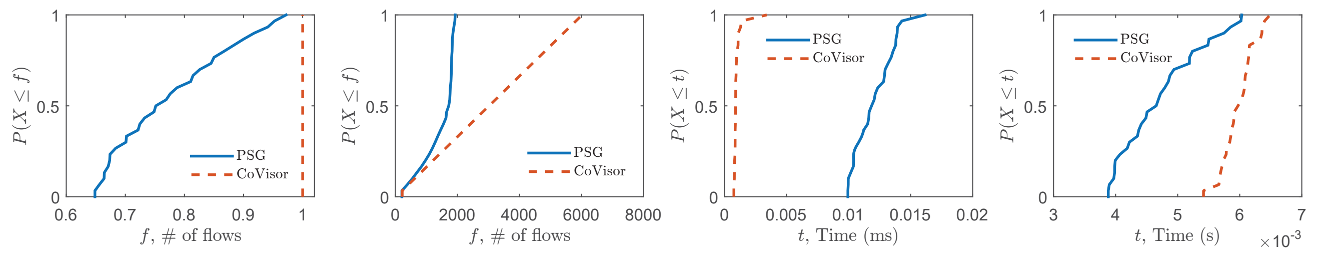
(a) Update length. (b) Total length. (c) Compile time. (d) Total time.

Fig. 4: Update performance for parallel composition.



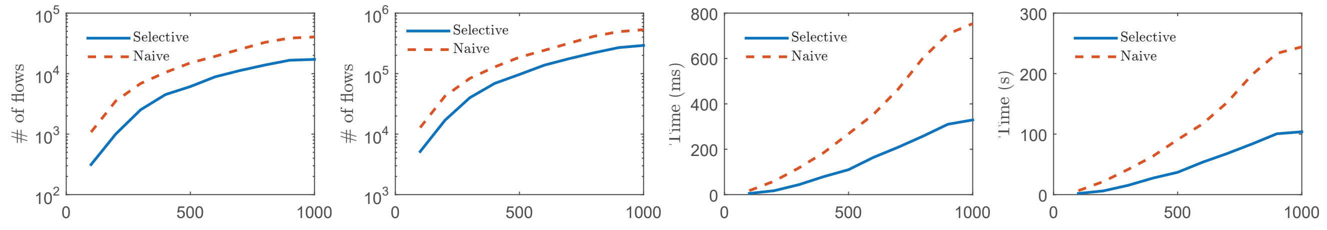
(a) Update length. (b) Total length. (c) Compile time. (d) Total time.

Fig. 5: Update performance for Sequential composition.



(a) Update length. (b) Total length. (c) Compile time. (d) Total time.

Fig. 6: Update performance for Override composition.



(a) Update length. (b) Total length. (c) Compile time. (d) Total time.

Fig. 7: Update performance for rule insertion into parallel composition.

C. *Rule Insertion*

In this section, we evaluate the performance of our method for translating insert operations for both single and composed policies. In the first experiment, we consider a single application policy and then randomly select a portion (2%, 5%, or fixed update length of 10) of rules and generate updates for inserting these rules. The base policy in this experiment is generated from firewall filter set by ClassBench [27]. We measure the update size for our insert translation, called *Selective* and the naive approach, called *Naive* for each process. As described in Section IV-E, an insertion request will be transformed into a set of add and delete rule requests. Thus, we consider the total number of these requests as the update length of insertion transformation. We also use the PSG composition approach for both cases which help us to eliminate anomalous rules. The process is repeated 100 times and then results are averaged.

Table I shows the average update length generated for insertions into a single policy. The results shows that the update generated by the Selective approach is often more than 10 order magnitude smaller than that of the Naive method.

In the second experiment, we evaluate the performance of rule insertion into a combined policy. To this end, we use the settings of the experiment for parallel composition in the previous section. Then, the updates is randomly generated by 10 rule insertion into the application policies. Fig. 7 shows the performance results for insertion into a parallel composition policy. As one can see in this figure, Selective

TABLE I: Average rule updates for inserting into a policy.

Base	2%		5%		Fixed=10	
Rules #	Naive	Selective	Naive	Selective	Naive	Selective
100	60.66	2.93	63.29	3.72	63.68	4.07
300	230.55	16.38	232.61	16.38	235.15	16.43
500	362.79	30.14	366.32	31.72	362.79	30.14
700	460.32	39.95	460.98	39.52	461.99	40.71
900	531.75	53.25	522.89	48.36	526.28	48.95
1100	614.15	60.87	559.30	52.08	585.98	62.40
1300	650.30	62.29	595.64	54.09	625.94	66.76
1500	695.12	64.10	610.86	57.22	657.91	67.46

usually generates updates with a length of around half of the length of updates generated by Naive. Moreover, the total length of composed policies generated by these two approach are approximately in the same order.

Figures 7(d) and 7(d) illustrate that both the compilation and total time of Naive considerably increase with the policy size, because larger policies force more updates including both deleting and adding rules. Since these updates need to be applied into the composition procedure, Naive is becoming much slower. On the other hand, Selective has a slight growth in both the compilation and total time, because it generates much less updates in transforming the insertion for the composition.

VI. Conclusions

In this paper, we proposed a novel framework for policy composition in a SDN hypervisor which considers both intra and inter anomaly detection and resolution. Moreover, we augmented the framework to efficiently transform priority change updates, such as rule insertion, in the application-level policies into the composed policy. We plan to extend the PSG model for bitmask feature proposed in the recent OpenFlow version. Moreover, supporting a network abstraction in PSG can help us to detect and resolve run-time anomalies in the network policies. We also plan to take advantage of multi-table support in OpenFlow to improve the performance of the policy composition.

Acknowledgment

This work was funded in part by the Google Faculty Award 2015 (Jha & Pagnucco).

References

[1] O. N. Foundation, "Software-defined networking: The new norm for networks," ONF White Paper, Tech. Rep., April 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[3] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, 2013, pp. 1–14.

[4] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 87–101.

[5] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular SDN languages," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, pp. 193–198.

[6] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*.

[7] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 199–213.

[8] H. Hu, G.-J. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 3, pp. 318–331, May 2012.

[9] M. Rezvani and R. Aryan, "Analyzing and resolving anomalies in firewall security policies based on propositional logic," in *IEEE 13th International Multi Topic Conference, INMIC*, 2009.

[10] R. Jalili and M. Rezvani, "Specification and verification of security policies in firewalls," in *Proceedings of the First EurAsian Conference on Information and Communication Technology*, 2002, pp. 154–163.

[11] P. Adao, C. Bozzato, G. D. Rossi, R. Focardi, and F. Luccio, "Mignis: A semantic based tool for firewall configuration," in *IEEE Computer Security Foundations Workshop*, July 2014, pp. 351 – 365.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[13] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," *SIGPLAN Not.*, vol. 49, no. 1, pp. 113–126, Jan. 2014.

[14] W. Han, H. Hu, and G.-J. Ahn, "LPM: Layered policy management for software-defined networks," in *Data and Applications Security and Privacy XXVIII*, ser. LNCS, 2014, vol. 8566, pp. 356–363.

[15] A. Dwaraki, S. Seetharaman, S. Natarajan, and T. Wolf, "GitFlow: Flow revision management for software-defined networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, 2015.

[16] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proceedings of the ISOC Network and Distributed System Security Symposium*, February 2013.

[17] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "PGA: Using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, pp. 29–42.

[18] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for NetKAT," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 328–341.

[19] "Openflow switch specification version 1.5.0," *https://www.opennetworking.org/*, 2014.

[20] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, ser. SafeConfig '10. New York, NY, USA: ACM, 2010, pp. 37–44.

[21] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13, 2013, pp. 99–112.

[22] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Elsevier/Morgan Kaufmann, 2005.

[23] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, 2012, pp. 49–54.

[24] C. Basile, A. Cappadonia, and A. Lioy, "Network-level access control policy analysis and transformation," *IEEE/ACM Trans. Netw.*, vol. 20, no. 4, pp. 985–998, Aug. 2012.

[25] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[26] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make your virtual SDNs programmable," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. ACM, 2014, pp. 25–30.

[27] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.