2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

An Adaptable Rule Placement for Software-Defined Networks

Shuyuan Zhang, Franjo Ivančić†, Cristian Lumezanu‡, Yifei Yuan§, Aarti Gupta‡, Sharad Malik**

* Princeton University, {shuyuanz, sharad@princeton.edu}
† Google Inc., {ivancic@google.com}
‡ NEC Laboratories America, {lume, agupta@nec-labs.com}
§ University of Pennsylvania, {yifeiy@cis.upenn.edu}

***Abstract*—There is a strong trend in networking to move towards Software-Defined Networks (SDN). SDNs enable easier network configuration through a separation between a centralized controller and a distributed dataplane comprising a network of switches. The controller implements network policies through installing rules on switches. Recently the "Big Switch" abstraction [1] was proposed as a specification mechanism for high-level network behavior, i.e., the network policies. The network operating system or compiler can use this specification for placing rules on individual switches. However, this is constrained by the limited capacity of the Ternary Content Addressable Memories (TCAMs) used for rules in each switch. We propose an Integer Linear Programming (ILP) based solution for placing rules on switches for a given firewall policy while optimizing for the total number of rules and meeting the switch capacity constraints. Experimental results demonstrate that our approach is scalable to practical sized networks.**

***Keywords*-SDN; Big Switch Abstraction; Rule Placement; Distributed Firewall**

I. Introduction

Cloud computing – the on-demand use of remote hardware or software computing resources over a network – has emerged as the main way to deploy new applications and services at scale. As the cloud requirements grow and the underlying data centers scale to ever more servers, the underlying network administration and management are becoming increasingly complex. This has led to the recent emergence of software-defined networking (SDN), which allows network operators to manage and control a network from a centralized server.

SDN separates the control plane of a network from the data plane. An operator can manage the physical functioning of the data plane through software installed on a general-purpose computer called the controller. The data plane comprises a network of switches responsible for line rate packet processing. The SDN controller communicates with the individual network switches through dedicated links and special-purpose control messages. The controller can thus install new routing rules on the switches, or delete rules etc. One popular SDN protocol that allows such communication between the controller and the network switches is OpenFlow [2] and there are many controller platforms that allow network administrators to write network applications [3], [4], [5], [6].

The recent emergence of SDN has encouraged the development of scalable rule management systems that allow treating the entire SDN network using the "Big Switch" abstraction [1]. This abstraction allows the operator to define end-to-end flow policies which are then compiled down to individual rules on different switches by a centralized rule management system. We consider such end-to-end flow policies to contain a number of different types of rules, including access control (firewall) rules, routing policy rules defining the paths taken by packets traversing the network, traffic shaping, or traffic engineering rules such as rules for load balancing, rules for fair accounting, etc.

In this paper, we focus on optimized automatic placement of access control list (ACL) or firewall rules based on a high-level ACL policy description and a given network topology. Each rule relates to a set of packets that are either permitted (PERMIT rule) or dropped (DROP rule). Rules related to routing of flows, i.e, which path is taken by packets, are considered to be managed and installed by a separate SDN controller module. *We consider how to optimally distribute the ACL rules among the switches, with the goal to optimize criteria like the total number of ACL rules installed, while satisfying per-switch capacity constraints for ACL purposes and maintaining all ACL policies.*

Recently, there have been a number of algorithms and heuristics proposed to address the centralized rule management system problem. These range from rule compression systems on single switches [7], [8], [9], [10], [11], [12], distributing ACL policies on network edge switches [13], distributing ACL policies while also changing routing on internal switches [14], as well as distributing ACL policies while respecting a separate routing module [1], [15]. Most approaches follow the idea of representing ACL policies using multi-dimensional packet (or flow) spaces, computing covers in some multi-dimensional packet space that corresponds to optimized rules, and distributing such covers over network paths. In the literature, various algorithms and heuristics have been suggested to compute such covers and handle the large space of possible network paths.

Rather than following the standard approach of considering ACL policies to be represented as multi-dimensional spaces, we formulate ACL policies as a prioritized rule list and propose a novel dependency graph based analysis. In this graph, each node represents a rule on a switch and the edges represent dependencies between the rules. These dependencies are based on permit-drop conflicts in the prioritized list of ACL policy rules. We then use this dependency graph to encode the rule placement problem as an optimization problem or a satisfiability problem, that can be solved by an ILP-solver (Integer-Linear Programming) or an SMT-solver (Satisfiability Modulo Theory) [16], respectively. (The satisfiability problem can also be solved by a Pseudo-Boolean solver [17].) To do so, we generate constraints based on the dependency graph that maintain the ACL policies, while taking the flow routing and switch capacities into account. Our encoding extends naturally to handle additional rule optimizations, e.g. merging rules at switches when possible. It also extends easily to different (and combinations of) objective criteria, e.g. total number of switches, distance from ingress of switches where rules are placed so as to minimize network traffic, weighted placement to favor certain switches, etc.

Our encoding is precise, i.e, the solver will not declare failure if a solution (satisfying or optimal, as required) exists. This is in contrast to previous approaches based on approximate abstractions, that may report failure even if a solution exists, especially in case of highly constrained instances. However, precisely solving the optimization problem may be complex and time-consuming. As such, it is suitable for finding an optimal solution when a new ACL policy takes effect, since such policy changes are rather infrequent. However, dynamic routing changes are more frequent, and they may impact previously placed ACL rules. In such situations, we want our ACL rule placement management system to quickly provide a satisfying solution, which need not be optimal. For this reason, we maintain both the optimization problem and the satisfiability problem (Section IV-E). Then, in a dynamically changing environment, we construct and solve the satisfiability problem for the smallest subset of switches impacted by the routing.

This paper makes the following contributions:

- It provides an ILP-based formulation for the rule-placement problem starting from (i) given ACL policies in the form of a prioritized set of rules and (ii) a given routing. The resulting solution, if one exists, meets the capacity constraints of individual switches and minimizes alternate possible objective functions such as the total number of rules or the estimated traffic by placing DROP rules further upstream.
- It extends this formulation to include the merging of rules common to multiple policies for greater rule minimization.
- It provides an alternative satisfiability-based formulation of the capacity constraint satisfaction problem, including allowing for merging, that can be solved by an SMT or a Pseudo-Boolean solver.

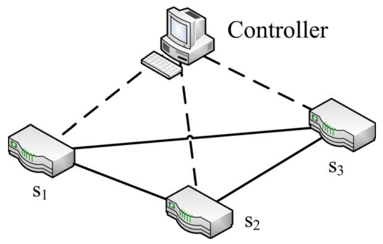


Figure 1. Software-Defined Network: Illustrative Sketch

- It provides an experimental evaluation that demonstrates the efficacy of the ILP formulation on practical sized networks.

The paper is organized as follows. Section II provides the relevant background on Software-Defined Networks. Section III formulates the rule placement problem that is the focus of this paper. In Section IV, we discuss our approach to this problem and describe how to use ILP and Satisfiability solvers (SMT and Pseudo-Boolean) for its solution. This is followed by an experimental evaluation of this method in Section V. We discuss relevant related work in Section VI and we provide concluding remarks in Section VII.

II. Background

A. *Software-Defined Networks*

SDN enables a separation between the control plane and the data plane by means of a centralized controller that controls every switch in the data plane [2]. Figure 1 shows an illustrative sketch of an SDN. Network policies are implemented by the controller through rules it installs and deletes on the switches through dedicated links shown in dotted lines in the figure. Each switch is relatively simple with several consecutive matching tables, with each matching table containing a strictly prioritized rule list. The rules in the matching table are composed of two fields, one matching field and one action field. The matching field specifies how the packet header will be matched against this rule by means of a match pattern consisting of an array of ternary elements, $\{0, 1, *\}$, where $*$ is a wildcard character that matches both 0 and 1. The packet header is matched against the rules in order of decreasing priority and is said to match the highest priority rule for which the matching field matches the header. The action field specifies the action to be taken on the matched packets, e.g. it can forward the packet to the next matching table or it may directly route the packets to the egress ports or the controller. The action field may also include packet header modifications, such as changing the VLAN (Virtual Local Area Network) tag or TTL (Time to Live) field. The controller can add and remove the rules in the matching tables through special messages. The switches can also send messages to the controller about their local events, such as packet mismatch, link status, or

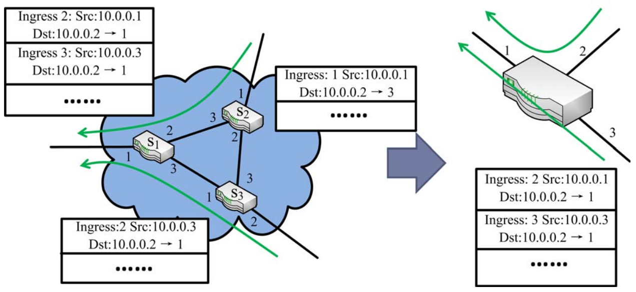


Figure 2. Big Switch Abstraction: Illustrative Sketch

rule information. For example, the switches may send all the packets that do not match any rules in the switch to the controller.

There are many advantages of using SDN. Since SDN switches are functionally simple and easy to design, the cost of switches is significantly lowered. Complicated network middleboxes can be replaced by cheap commodity hardware to obtain the same functionality. Further, SDN enables extremely flexible network management. For example, networks can be easily virtualized using SDN [18] and this is very valuable for networks of multi-tenant datacenters.

B. *Big Switch Abstraction*

Recently, researchers have introduced the concept of a "Big Switch" abstraction [1] as a network specification mechanism. Since the core function of a network is packet forwarding, we can model the network as a virtual "Big Switch" as long as we have a description of the network's behavior (Figure 2). This behavior can be defined by two kinds of policies:

- a routing policy to define how packets will traverse the network from an ingress to an egress
- an endpoint policy to describe other aspects of the network, such as access control, packet monitoring, accounting, etc.

These two policies may change dynamically as the network operates and therefore, the state of the abstract switch may also change. A key advantage of abstracting the network as one switch is that it helps relieve the burden of configuring the network. Instead of managing the network, switch by switch in the controller, the network administrators only need to specify the high-level behavior for the network while leaving the conversion of the "Big Switch" abstraction to the low-level rule placement to be done by the network operating system or network compiler.

The conversion from a high-level description of the network into a detailed rule placement on each switch is challenging because many constraints have to be satisfied to achieve a correct and efficient network implementation. These challenges are summarized below.

- The results obtained from the rule placement algorithm have to preserve the semantics of the original policies. Since the rules are prioritized, a rule of lower priority exists in the context of rules of higher priority, i.e. a packet header can match a rule only if it does not match any rules of higher priority. Thus, a lower priority rule cannot be placed without considering its relationship to higher priority rules. Thus, to maintain the priority semantics, we have to create a dependency relationship between rules and account for this during rule placement.
- The rule placement is constrained by the rule capacity limitation in each switch. The main component inside an SDN switch (or OpenFlow supported switch) is a Ternary Content Addressable Memory (TCAM) that provides parallel packet matching. However, a TCAM is very expensive in terms of the power consumption and thus is a scarce resource for each switch. This limits the number of rules per switch. The size of the TCAM is usually $1k \sim 2k$ ($1.5k$ TCAM slots in the 5406zl switch [19]).
- The rule placement algorithm may impact network traffic, and thus we need to ensure that any rule placement suitably trades off optimizing the number of rules with the possible increase in network traffic.
- There may be multiple rule placements that meet the capacity limitation of all switches. Determining an optimal solution for a given objective function involves searching a possibly large number of placement alternatives.

C. *Distributed Firewalls*

As part of the endpoint policy, the access control list (ACL) is critical to the security of the network. Many cloud computing providers offer a way to have user-specified distributed firewall policies, such as Google Compute Engine's firewall policy [20]. Google Compute Engine allows users to specify one firewall policy to be associated with each instance created in the cloud and all packets flowing from and to the instance are subject to this firewall policy.

Most firewall policies can be modeled as a list of priority rules. This is compatible with the matching tables for OpenFlow switches. As with OpenFlow's matching rules, each ACL rule is a tuple of a matching field and a binary decision field which serves as the action field. The decision field has one bit to specify if this rule drops or permits the packet (this is also supported by OpenFlow).

III. Problem Definition

In this paper, we are mainly concerned with how to place a distributed firewall in the network. There is some flexibility in how rules are placed in switches to accomplish a given firewall policy. Thus, this flexibility can be exploited to try

Table I
PROBLEM FORMULATION: NOTATION

N	The set of switches in the network
s_i	Switch i
C_i	The capacity of switch i
l_i	Network entry (ingress and egress) port i
P_i	The set of paths originating from the network ingress i
$p_{i,j}$	$p_{i,j} \in P_i, p_{i,j} = \{s_x, s_y ...\}$ and it represents the set of switches on path j
S_i	$S_i = \bigcup_j p_{i,j}$ and it represents the set of switches reachable from l_i
Q_i	The policy attached to ingress i
$r_{i,j}$	$r_{i,j}$ is a single rule and $r_{i,j} \in Q_i$
$m_{i,j}$	The matching field of $r_{i,j}$
$d_{i,j}$	The decision of the rule
$t_{i,j}$	The priority of the rule

and optimize the number of rules or alternate objectives while meeting the rule capacity of each switch.

The relevant notation used in the problem formulation is listed in Table I.

The network N is composed of a set of switches and each switch is named using its index, thus the i^{th} switch is s_i with capacity C_i. Each switch has a set of ports, and switches are connected to each other using these ports. Some of the ports may be the entry/exit points for the network and we use l_i to represent the i^{th} network ingress/egress port.

We assume that the routing policy is generated by some external module. The routing module may run shortest-path routing to generate the paths and it may consider certain load balancing schemes under different traffic patterns. Or it may simply be a static routing library. The design of this module is beyond the scope of this work. We only require that the routes obtained from the routing policy are provided as an input to this problem as a set of routing paths. If l_i is an ingress port, then P_i is the set of paths originating from l_i. Each path in P_i is an ordered set of switches, $p_{i,j}$. S_i is the aggregation of all the switches in all the paths originating from port l_i.

The distributed firewall policy is specified as a set of firewall policies $\{Q_i\}$, one for each ingress port. This is similar in form to Google Compute Engine's firewall policy [20] and Amazon's EC2 security group [21]. Each Q_i is composed of a set of rules $r_{i,j}$. Each rule $r_{i,j}$ is a tuple $(m_{i,j}, d_{i,j}, t_{i,j})$, where

- m is an array of three-valued elements specifying the matching field ($\{0, 1, *\}$ where $*$ is a wildcard that matches both 0 and 1)
- d is the action, which can be either DROP or PERMIT
- t encodes the priority of each rule

Policy rules are strictly prioritized, i.e, $\forall i \forall j \forall k (j \neq k)$: $t_{i,j} \neq t_{i,k}$. If $t_{i,j} < t_{i,k}$, rule $r_{i,j}$ has a lower priority than $r_{i,k}$.

The rule placement problem can be defined as follows: Given N, P, Q, generate a mapping from $r_{i,j}$ to S_i, i.e. assign the rule $r_{i,j}$ to one or more switches $s_k, s_l, s_m, \ldots$

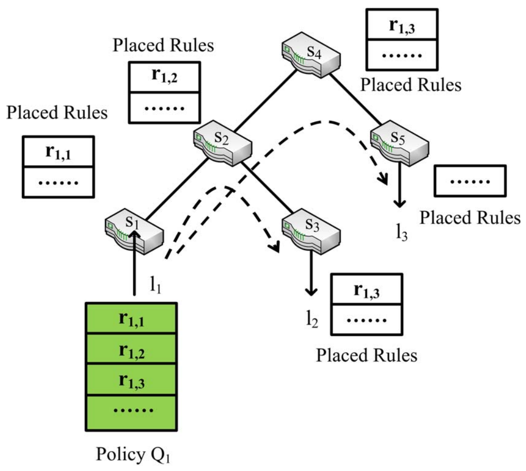


Figure 3. Problem Illustration

reachable from l_i, such that the total number of rules placed in switch s_k does not exceed its capacity C_k. Note that since a rule may be placed in more that one switch, different placements will lead to a different total number of rules. As we will see, this basic formulation can be augmented with alternative objective functions.

This is illustrated through a small example shown in Figure 3. Switches s_3, s_5 have a network egress port and all packets from the ingress l_1 are destined to either l_2 or l_3. The routing module specifies that packets have to go through s_1, s_2, s_3 to reach l_2, and through s_1, s_2, s_4, s_5 to reach l_3 (dotted lines in the figure). The policy, Q_1, attached to the ingress at l_1 is shown in the Figure. Then, the rule placement problem is to find a valid mapping of the rules to the switches such that all packets destined to l_2 have to match the complete policy Q_1 along its path of s_1, s_2, s_3 without violating the capacity constraint of any of the switches. Similarly for packets destined to l_3. The solution for this example shows that rule $r_{1,3}$ is replicated along both paths when it is placed on both s_3 and s_5.

Ideally all ACL rules should be placed on the ingress switches. This would lead to the least network traffic. Also, if all rules are not placed on the ingress switch, then this can lead to increased TCAM space usage as the rules are possibly replicated along different paths from the ingress switch, as with $r_{1,3}$ in the example above. Since capacity constraints will likely prohibit all rules from being placed in the ingress switch, and it is difficult to manually explore the different options that satisfy the various capacity and other rule-placement constraints, there is need for an automated solution for the rule placement problem.

Another solution is to put all the firewall policies in the end hosts, meaning running software such as Linux iptables [22] or Open vSwitch [23] on the host machine's operating system or hypervisor. However, software-based solutions are generally slower than TCAM-based hardware solutions ones because it is difficult for software to match thousands of rules simultaneously. Further, in a virtualized cloud environment, occupying sigfinicant system resources can degrade the tenant application's performance.

Real-time or online placement may be needed in response to small policy changes, e.g. in response to security related updates where large latencies may not be acceptable. While it may be possible to make these changes through an ad-hoc method, it is desirable to have an analytical solution framework which allows for an incremental solution that can run in a fraction of a second to a second (depending on the magnitude of the change) and thus be able to provide solutions where a non-analytical method may not be able to. This notion of real-time changes is consistent with that used in other similar contexts, e.g. in incremental data-plane verification in SDNs using real-time Header Space Analysis [24] which can take up to seconds for real-time verification for link updates.

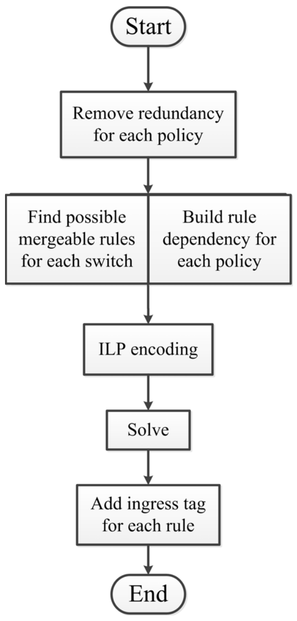
IV. Solution Approach

In this work, we transform the rule placement problem into an Integer Linear Programming (ILP) problem, which satisfies the switch capacity, priority, and policy constraints and optimizes some objective function such as minimizing the total number of rules. We also formulate a satisfiability problem which meets the above constraints without the objective optimization. This satisfiability problem can be solved using SMT or Pseudo-Boolean Solvers. A flow chart for our approach is shown in Figure 4. We start with an optional stage of removing the redundant rules in each policy by using techniques proposed in [7], [8], [9]. Then, we build the rule dependency graph (explained in Section IV-A1) and find mergeable rules (Section IV-B). These steps are independent and can be done in any order. The next two steps handle the ILP formulation and solution, and finally, we add ingress tags to each rule (Section IV-A5).

The basic variable of the ILP formulation is the binary variable, $v_{i,j,k}$, which indicates if rule $r_{i,j}$ is placed on s_k, where $s_k \in S_i$. $v_{i,j,k} = 1$ if and only if $r_{i,j}$ is placed on s_k. A rule may be placed on more than one switch, e.g. it may be placed on two different switches along two different paths in P_i. Note that we do not construct new rules or modify rules (besides merging as discussed below). Next, we discuss the constraints on the v variables and the objective function in the ILP formulation.

A. ILP Formulation

Besides being binary $(0, 1)$, the v variables are constrained in other ways. The Rule Dependency Constraints ensure that the semantics of the prioritization in the ACL rules is obeyed during placement. The Path Dependency Constraints ensure that every drop rule in a policy has to be placed somewhere along every path from an ingress. The Switch Capacity Constraints, constrain the number of rules in each switch based on its capacity.



Start
Remove redundancy for each policy
Find possible mergeable rules for each switch
Build rule dependency for each policy
ILP encoding
Solve
Add ingress tag for each rule
End

Figure 4. Flow Chart for Our Approach

1) Rule Dependency Constraint: The deployed distributed firewalls have to drop exactly the packets as specified in the given policy, i.e., the deployed policy should not drop a packet which is specified as PERMIT and it should drop every packet specified as DROP. Since the firewall has a binary action, the DROP rules are complementary to the PERMIT rules and thus, we only consider DROP rules. (An alternative formulation that considers only PERMIT rules would be similar.) Thus, the problem reduces to how to place the DROP rules on every path originating from an ingress.

To place DROP rule $r_{i,j}$ in a switch, we need to understand which other rules are affected by such a placement.

- Rules with matching fields disjoint with $r_{i,j}$ are not affected because they work on a different set of packets.
- Other DROP rules, even with an overlapping, i.e. non-disjoint, matching field are not affected because they have the same action as $r_{i,j}$. Placement of $r_{i,j}$ on a switch does not pose any constraints on where other DROP rules shall be located because it does not matter where to drop the packet as long as it is dropped.
- PERMIT rules with higher priority and an overlapping matching field are placed in the same switch as we place $r_{i,j}$. If a PERMIT rule intersects with $r_{i,j}$, then it potentially permits some packet that may also match the lower priority DROP rule $r_{i,j}$. Thus, this PERMIT rule must be placed as a higher priority rule on the same switch as $r_{i,j}$ to ensure that the permitted packets continue to be permitted and are not dropped by $r_{i,j}$.

We use a third subscript on variables $m_{i,j,k}, d_{i,j,k}, t_{i,j,k}$ to indicate which switch we actually place $r_{i,j}$. This rule dependency constraint is encoded as follows:

$$\forall i \forall k \forall u, w(d_{i,u,k} = PERMIT, d_{i,w,k} = DROP,$$
$$m_{i,u,k} \cap m_{i,w,k} \neq \emptyset, t_{i,u,k} > t_{i,w,k}) : v_{i,u,k} \geq v_{i,w,k} \quad (1)$$

Since v is binary, this formula ensures that if DROP rule $r_{i,w}$ is placed on switch s_k, PERMIT rule $r_{i,u}$ has to be placed on that switch as well.

2) Path Dependency Constraint: Every DROP rule has to be placed somewhere along each path from the ingress. (We defer the discussion on redundant rules to Section IV-C, and assume that there are no redundant rules here.) This constraint is expressed as follows:

$$\forall r_{i,j}(d_{i,j} = DROP) : (\sum_{s_l \in S_i} v_{i,j,l}) \geq 1 \quad (2)$$

3) Switch Capacity Constraint: Further, the number of rules placed on a switch must not exceed the number of rules the switch can hold. This constraint is specified as:

$$\forall k : (\sum_{i,j} v_{i,j,k}) \leq C_k \quad (3)$$

4) Objective Function: In addition to satisfying constraints, an ILP formulation allows for optimizing an objective function. For example, we can minimize the total number of rules in the network:

$$\sum_{i,j,k} v_{i,j,k}$$

This maximizes the available rule space for the future addition of rules. Alternatively, we can optimize the location of the drop rules such that they are placed further upstream along the path to minimize the traffic induced by the rule placement. Then, the objective function is to minimize:

$$\sum_{i,j,k} (v_{i,j,k} \times loc(s_k, P_i))$$

where the function $loc(s_k, P_i)$ calculates the distance (number of hops) between an ingress port and the switch s_k and it can be determined in the compile time.

5) Identifying Ingress Policy using Tags: Since switches contain rules for different ingress port policies, Q_i, we need to have a mechanism to identify which policy a rule applies to. One solution is to add an ingress-policy-specific tagging field, such as VLAN tag. Whenever a packet enters the network, the VLAN field is used to indicate the ingress port where it enters the network. This tag then has to be included as a part of the matching field. The priority of rules deployed in each switch has to respect the original priority in the policy. However, the relative orders of the rules from different ingress policies does not matter because the tagging field results in a non-overlapping rule space.

B. Rule Merging

Networks often have a network-wide blacklisting policy, which specifies that no packet from any port shall go to certain places, or packets originating from certain IPs are potentially dangerous and need to be dropped. To take advantage of the fact that some rules are global to every ingress policy, we can merge certain rules across policies to further reduce the rule space required. The merging rules are merged if they are identical, i.e., the rules have the same matching field and the same action but belong to different policies. When rules are merged, the resulting tagging field is the union of the policies of the merged rules.

We now present the encoding for merging identical rules. We use the binary variable $v^m_{i,j}$ to represent a possible merged rule i at switch s_j. The superscript "m" means "mergeable". This rule depends on a set of other rules from the ingress policies, $R^m_{i,j} = \{v_{a,x,j}, v_{b,y,j}, ...\}$. Note that we have represented the rules here by variables indicating their presence or absence. $R^m_{i,j}$ captures the set of rules from all the paths traversing switch s_j that are identical except for the policy they apply to. $v^m_{i,j}$ is 1 if and only if $\forall v \in R^m_{i,j} : v = 1$. This constraint can be expressed as:

$$v^m_{i,j} \geq (\sum_{v \in R^m_{i,j}} v) - (M - 1) \quad (4)$$

$$v^m_{i,j} \leq \frac{1}{M} \sum_{v \in R^m_{i,j}} v \quad (5)$$

where $M = |R^m_{i,j}|$. From equation 4, if all of $v \in R^m$ are equal to 1, $v^m_{i,j} \geq (M) - (M-1) = 1$, i.e., it is equal to 1. If there exists $v \in R^m_{i,j}$ such that $v \neq 1$, Equation 5 specifies that $v^m_{i,j} \leq \frac{x}{M} < 1$, i.e., $v^m_{i,j} = 0$.

There is a corresponding change in the rule capacity constraint and objective function. The set of rule placement variables now includes $v^m_{i,j}$, and each $v \in R^m_{i,j}$ is replaced by $v - v^m_{i,j}$. Thus, if $v^m_{i,j}$ is 0, then each $v \in R^m_{i,j}$ is reflected in the capacity constraint and the objective function as in the non-merged case. If $v^m_{i,j}$ is 1, then each $v \in R^m_{i,j}$ does not contribute to the capacity constraint and the objective function, and all $v \in R^m_{i,j}$ are effectively replaced by a single $v^m_{i,j}$.

There is one subtle issue that needs to be addressed during rule merging that has to do with rule dependencies arising from rule priorities. It is possible that there exists a circular dependency between the mergeable rules as illustrated by the following example. The core of this example is two rules, with rule 1 having higher priority than rule 2 in some policies and a lower priority than rule 2 in others. Further, there are three paths A, B, and C traversing a switch. All three paths have these two rules in their ingress port policies, one of which is a permit rule r_1 with matching field of src:10.0.0.0/16 and dst:11.0.0.0/8. The other rule is a drop rule r_2 with matching field of src:10.0.0.0/8 and

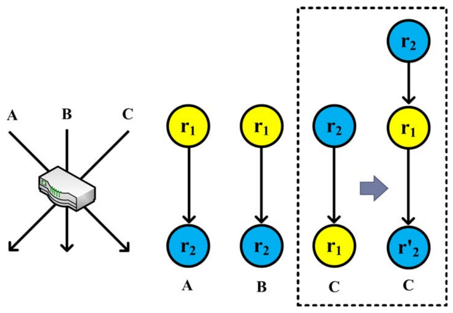
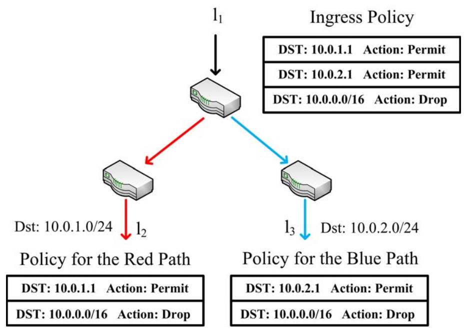


Figure 5. Circular Dependency for Mergeable Rules


l1
Ingress Policy
DST: 10.0.1.1 Action: Permit
DST: 10.0.2.1 Action: Permit
DST: 10.0.0.0/16 Action: Drop
Dst: 10.0.1.0/24
l2
l3
Dst: 10.0.2.0/24
Policy for the Red Path
DST: 10.0.1.1 Action: Permit
DST: 10.0.0.0/16 Action: Drop
Policy for the Blue Path
DST: 10.0.2.1 Action: Permit
DST: 10.0.0.0/16 Action: Drop
Figure 6. Sub-policy for each route

dst:11.0.0.0/16. The dependency relationship for all three paths is shown in Figure 5. If we decide to share the three rules, there will be a circular dependency since we have to put r_1 before r_2 for path A and B but the order is reversed for path C; otherwise, the semantics of the policies will not be obeyed. Hence, the dependency relation cannot be fulfilled if we do not break this circular dependency.

This circular dependency can be easily broken by adding dummy rules to the policy. In the previous example, we can add a dummy rule r_2' to the policy of path C, which is exactly the same as r_2 but has a lower priority than r_1. Then, we mark r_2 as not mergeable and we merge rule r_2'. This technique does not change the semantics of the policies because r_2' is dominated by r_2 and can never by matched.

C. Path-Sliced Policy Rules

Often, the routing library not only specifies all the packet routes, but also the flows or packets traversing each route. It is possible that the set of packets following a route only matches a portion of the rules of the ingress policy. Instead of placing all rules of the policy onto the route, we only need to place the rules that are overlapping with the packets and discard the non-overlapping rules. Figure 6 shows an example. Assume the ingress policy has three rules and there are two routes originating from the ingress. Packets for one route are destined to IP prefix 10.0.1.0/24 and packets for the other route are destined to prefix 10.0.2.0/24. Since packets for the red route only match the first and the third rules, we only need to place these two rules onto the route, instead of placing all three of them. Similarly, only the second and the third rules need to be placed for the blue route. This path-specific slicing of policy rules is managed by limiting the path dependency constraint (Equation 2) to rules that need to be placed for the path

D. Satisfiability Encoding

If we are interested in a solution that satisfies the three sets of constraints and do not need to optimize any objective function, then this problem can be encoded as a satisfiability problem that can be solved using an SMT or Pseudo-Boolean Solver. In this formulation, $v_{i,j,k}$ is a Boolean variable or a binary integer variable depending on the constraint. This is a slight abuse of notation that simplifies the formulation. Equation 1 is expressed as:

$$\forall i \forall k \forall u, w(d_{i,u,k} = PERMIT, d_{i,w,k} = DROP,$$
$$m_{i,u,k} \cap m_{i,w,k} \neq \emptyset, t_{i,u,k} > t_{i,w,k}) : \bigwedge(v_{i,w,k} \rightarrow v_{i,u,k}) \tag{6}$$

Here $\bigwedge$ is the logical AND operation. Equation 2 is expressed as:

$$\forall r_{i,j}(d_{i,j} = DROP) : \bigvee_{s_l \in S_i} v_{i,j,l} \tag{7}$$

Here $\bigvee$ is the logical OR operation. Equation 3 stays the same with $v_{i,j,k}$ being a binary integer variable. The equations 4 and 5 for merging rules reduce to the following constraint:

$$v_{i,j}^m = \bigwedge_{v \in R_{i,j}^m} v \tag{8}$$

which specifies that v^m is 1 if and only if all v are 1. All constraints need to be satisfied to achieve a satisfiable solution.

E. Incremental Deployment

The network is a dynamic system and the packet routes may change from time to time, requiring our placement to fast re-adapt to the new routing policies. The network changes include the following [1]:

1) **Ingress Policy Change:** Rules inside a policy can be modified or removed and new rules can be added to each policy.
2) **Ingress Policy Installation**: When new switches join the network, new policies may need to be placed in the network.
3) **Routing Policy Change:** New routes may be added or old routes deleted from the network.

The network update can also be at different scales:

1) **Small Scale:** Only a small part of the network is changed such as a new route is added or only a few rules for a single policy are changed.

2) **Medium Scale:** A portion of routes are updated or multiple policies need to be modified. This can happen, for example, when tenants leave or join a datacenter.
3) **Large Scale:** The majority of the network needs to be updated. This can happen in planned network upgrade or planned maintenance.

As seen in the experimental evaluation, running the ILP solver can take from a fraction of a second to a hundred seconds or so. This may be impractical for small and medium scale updates, which may require shorter latencies, e.g. for security updates needing rapid deployment. This prompts a look at alternative solutions for these cases.

Similar to the techniques used by [1], if the update scale is small, it may be possible to find a solution by modifying the existing solution using some heuristic. For example, if a new rule is added to the policy, we can try to place the rules as close to the ingress as possible. Such a simple heuristic may be enough to obtain a satisfying solution. While it may not guarantee a globally optimal one, it is a good trade-off between performance and optimality.

For medium scale updates, there may be an advantage in constructing a sub-problem that is smaller than the original problem by limiting the constraints and the objective function to the policies, paths and switches impacted by the change. Consider the case where an old route is removed and a new route added. We remove the rules corresponding to the old route and add variables corresponding to the new route. All other rule placements are fixed as in the original solution. This incremental version is then solved. This version is restrictive as it does not allow any rules not relating to these paths to change. Thus, it is possible it may have no solution that satisfies all constraints, even though solving the problem from scratch may have resulted in a feasible solution. The hope is that if the changes are small, the solver may be able to modify the original solution to meet all constraints. Similarly, even if all constraints are met, the solution may be sub-optimal. Since in the context of the rule placement problem, the constraints are more critical and optimality is an optional desired goal, this may be less of an issue.

V. Experimental Results

We tested the effectiveness of our approach through a set of experiments using an ILP-based rule-placement tool we developed based on the flow in Figure 4. (The experimental evaluation of the satisfiability based formulation in Section IV-D is the subject of future work.) The primary goal of these experiments was to study the scalability of the proposed technique. As there is no other approach that accomplishes rule placement with global network-level optimization across multiple paths and multiple policies, it was not possible to do a direct comparison with an alternative method.

We used CPLEX as the underlying ILP solver [25]. The benchmarks used were synthetically generated. This is standard in the networking community where proprietary network data is hard to obtain. Further, the synthetic benchmarks permit for constructing a family of benchmarks that can be used in scalability studies. The underlying network topology used was a Fat-Tree graph [26]. The policy for each network ingress was generated by ClassBench [27], a synthetic firewall benchmark generator. A randomly generated shortest-path routing was used as the routing policy. The objective function was to minimize the total number of rules inside the network. As mentioned earlier, this maximizes the slack available for adding rules in the future. The computing equipment used in these experiments had an Intel Xeon processor (3.2 GHz) running Linux (kernel version 3.2.0).

We ran five experiments as follows to demonstrate the scalability of the approach.

- **Experiment 1:** For the first set of experiments, we fixed the topology, the routing paths, and the switch capacity while increasing the number of rules n at each ingress (Figure 7, Figure 8 and Figure 9). For each of these cases, the numbers of paths p in the network is 1024, and the number of rules n ranges from 20 to 110. This is based on practical sized policies presented in the literature [28]. The y-axis plots the run-time (log) for the rule placement procedure. As the policies are randomly generated, there is a variation in runtimes depending on the specific instance generated. To account for this, we ran 5 instances for each point along the x-axis (# rules in the policy) and indicate the average run-time with variation bars along the y-axis. Note that for a Fat-Tree topology, the number of switches is equal to $5k^2/4$ and the number of hosts is $k^3/4$, where k is the number of ports per switch [26]. The three figures correspond to different network sizes specified in terms of k. For each k, we consider two switch capacities, C, 200 and 1000. Again, this represents the possible range of interest in practical switches, e.g., there are $1.5k$ TCAM slots in the 5406zl switch [19]. Only a fraction of these will be available for ACL rules, with the rest used for routing. We make the following observations. (i) For a given network size (k), the runtime is higher for the smaller switch capacity ($C = 200$) as the instance is more tightly constrained. (ii) The runtime generally increases with the number of rules except when the instance is over-constrained (no solution exists) with a large number of rules. The cases ($k = 8, C = 200, r = 90, 100, 110$), ($k = 16, C = 200, r = 110$), and ($k = 32, C = 200, r = 110$, except one data point) are infeasible and all others return the optimal solution. For the case with $k = 32, C = 200, r = 110$, one data point took 26 minutes to return the optimal value

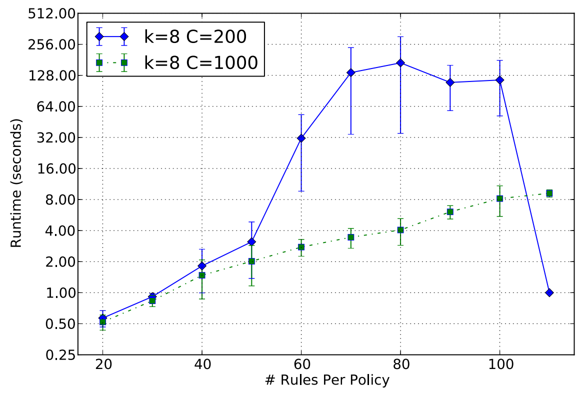


Figure 7. Execution time vs. # of Rules. $k = 8$

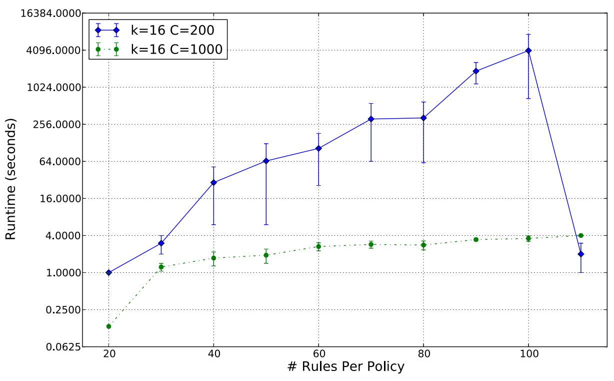


Figure 8. Execution time vs. # of Rules. $k = 16$

and all the rest return infeasible within 20 seconds. The execution time for the infeasible cases actually decreases and there is a sudden drop when r transitions from 100 to 110 in all three figures. At this transition the problem instances become over-constrained, and the solver discovers that and returns quickly. Similarly, if the problem is under-constrained, as when the capacity of the switches is large (such as $C = 1000$), and it is easy to find a solution, the execution time is quite short.

- **Experiment 2:** For the second set of experiments, we fixed the topology and the total number of rules while increasing the number of paths from 256 to 2048 with a step of 256 (Figure 10). For this set of experiments, we used a network with $k = 8$, $r = 100$, and $C = 200, 500$. With $C = 200$, the solver returns infeasible when $p > 512$. The run times indicate both easy and difficult to prove infeasible cases. With $C = 500$, all points are feasible and the execution time is flat indicating that the number of paths is not as significant as the number of rules or the number of

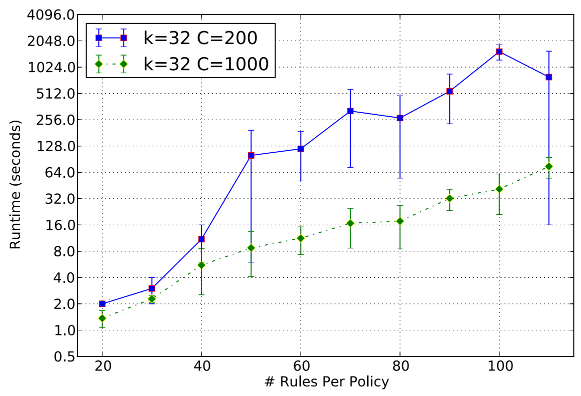


Figure 9. Execution time vs. # of Rules. $k = 32$

switches if the switches are not capacity constrained.

- **Experiment 3:** In this experiment we studied rule merging for a network with $k = 8$, $p = 1024$. The number of rules that are not mergeable is 20 while the number of mergeable rules increases from 1 to 10. The capacity of the switches grows from 65 to 75. The results are shown in Table II. The columns "65-MR", "70-MR", and "75-MR" represent the cases when rule merging was enabled for the three different capacities and the columns without the "-MR" suffix indicate the cases when rule merging was not enabled. In each column, the left sub-column is the total number of rules installed in the network and the right sub-column reports the percentage overhead due to the rules not being able to fit on the ingress switch and the consequent duplication of rules over paths. "Inf" (infeasible) indicates that the solution was infeasible due to a capacity constraint. Let A be the total number of rules on all policies. If these rules could all fit in the ingress switches, then A would be the total number of rules in the network. However, when this is not possible, as rules are spread across paths, they may need to be duplicated along different paths. Let the total number of resulting rules placed in the network be B (reported in the left-subcolumn). B is larger than A due to rule duplication. The overhead of this duplication is given by $(B - A)/A$ which is reported as a percentage in the right-subcolumn of the table.
We make the following observations. (i) Rule merging results in several infeasible cases becoming feasible as the merged rules can meet the capacity constraints. (ii) Overall, the overhead of rule duplication is reduced by an average of 15% by rule merging. (iii) In some cases, the overhead may become negative. This reflects the savings due to merging across different policies.
- **Experiment 4:** In this experiment we study the effect

Table II
CAPACITY VS. OVERHEAD IN RULE MERGING

# MR	Capacity (with and without merging)											
Rules	65		65-MR		70		70-MR		75		75-MR	
1	3.8k	41%	3.5k	30%	3.5k	30%	3.3k	21%	3.1k	16%	2.9k	9%
2	-	Inf	3.6k	29%	3.9k	37%	3.3k	18%	3.6k	26%	3.1k	8%
3	-	Inf	3.6k	23%	4.2k	43%	3.6k	24%	3.9k	33%	3.1k	6%
4	-	Inf	3.7k	21%	-	Inf	3.4k	11%	4.3k	40%	3.1k	10%
5	-	Inf	3.8k	19%	-	Inf	3.9k	22%	4.7k	45%	3.9k	22%
6	-	Inf	-	Inf	-	Inf	3.6k	8%	-	Inf	3.3k	-1%
7	-	Inf	-	Inf	-	Inf	3.7k	6%	-	Inf	3.3k	-4%
8	-	Inf	-	Inf	-	Inf	3.7k	4%	-	Inf	3.4k	-6%
9	-	Inf	-	Inf	-	Inf	3.8k	3%	-	Inf	3.5k	-5%
10	-	Inf	-	Inf	-	Inf	4.9k	26%	-	Inf	3.5k	-9%

of changing switch capacity. We increase the switch capacity from 50 to 1000 for a fixed network with $k = 16$, $r = 100$, and $p = 1024$ (Figure 11). CPLEX returns infeasible quickly for all data of $C = 50, 100$. We make the following observations: As the capacity becomes larger, the execution time rises and then dramatically decreases. The data points in the tail have a lower execution time and a very small variance. This indicates that the under-constrained and over-constrained cases are relatively easier to solve.

- **Experiment 5:** In this experiment, we studied the feasibility of adapting the solution incrementally for new policy installation. Since rule deletion is relatively easy and rule modification can be modelled as a combination of rule deletion and installation, the execution time for rule installation is of more interest to us. We first ran the algorithm using a network with $k = 16, r = 100, p = 1024, C = 500$ and determined the spare rule capacity for each switch. Then we use this as the new capacity specification for incremental rule placement. For the experiment we considered adding $64, 128$, and 256 new policies with each policy having 100 rules and a single path to place the rules for each policy. Even though it is a single path for the policy, the rule dependency and capacity constraints still need to be met. All three cases finished within 1.2 seconds. The cases with 64 and 128 returned a feasible solution, and 256 returned infeasible. For the second part of the experiment, we still use the spare rule capacity for each switch but instead of adding more policies, we modify the original policies by forcing them to be placed in fewer or more paths and this captures a routing path change. We modified 1, 16, and 32 policies and they finish in 126, 217, and 442 ms respectively. This demonstrates that small policy changes can be handled relatively quickly compared to a complete solution starting from scratch.

The ILP instances generated by theses experiments could all be handled by the CPLEX solver. The total number of variables is proportional to the total number of rules and switches. The number of constraints is proportional to the number of paths, switches, and correlated with the number of rules (dependency constraints). For the case with $k = 8, r = 100, p = 1024$, we have about 290K variables and 520K constraints. It takes about 2 minutes to return infeasible for $C = 200$ and 8 seconds to return optimal for $C = 1000$. For the case with $k = 32, r = 100, p = 1024$, we have about 500K variables and 940K constraints. It takes about half an hour for $C = 200$ and about 12 seconds for $C = 1000$. In both cases we can obtain the optimal solution. Thus, the run time is acceptable for the scale of problems of interest. Further, this long computation time is only for the initial configuration and we use the incremental deployment for all the later network changes. This only takes a fraction of a second and hence can be used in online or real-time deployment.

Further, the technique implicitly shares rules across paths by placing them at switches common to many paths whenever permitted by switch capacities. This is in contrast to other techniques which place all rules in all paths and thus end up placing $p \times r$ rules in the network [1]. These techniques have a significantly higher rule placement overhead compared to our modest overhead seen in the results of Table II. For example, the total number of rules deployed for the largest overhead case (p=1024 and r=25 without rule merging enabled at row 5, column 75) is 4650, which is only 18% of $p \times r = 25k$. Also, our approach does not preclude the greedy solution, which places all the rules in the ingress switches as long as this solution is satisfiable and optimal in terms of the objective function.

In summary, the experiments demonstrate (i) scalability, as the ILP based solution can be applied to practical sized networks with acceptable run time (ii) good quality of results, as measured by the low overhead of rule duplication resulting from spreading the rules across paths.

VI. RELATED WORK & DISCUSSION

The works that are closest to our approach are [1], [15] and [14]. Palette [15] tries to distribute a global policy to every route in the network by partitioning the whole table into small pieces and placing each piece in one switch. Their formulation of the problem is much simpler than ours because they are limited to the same policy for every path in the network, which may not be the case in multi-tenant datacenters. Kang et al. [1] formulated the placement problem as distributing a global policy, specified as a union of one policy per route, to the network. However, their work did not leverage the fact that many rules can be merged together if they share the same origin. Further, they did not make use of the possibility of sharing the network-wide blacklisting rules. vCRIB [14] tried to balance the rule space between the physical networks and the hypervisors in the servers. One assumption they made is that they are allowed to change the packet route to achieve a feasible solution.

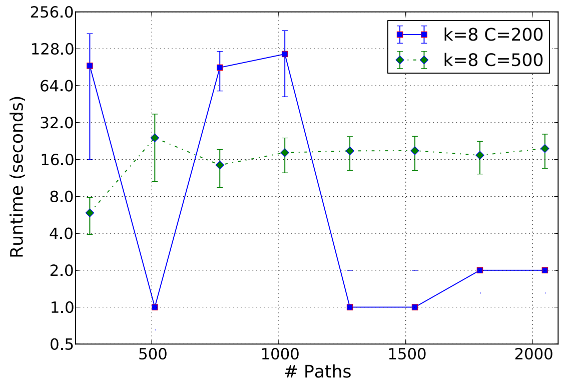


Figure 10. Execution time vs # Paths .

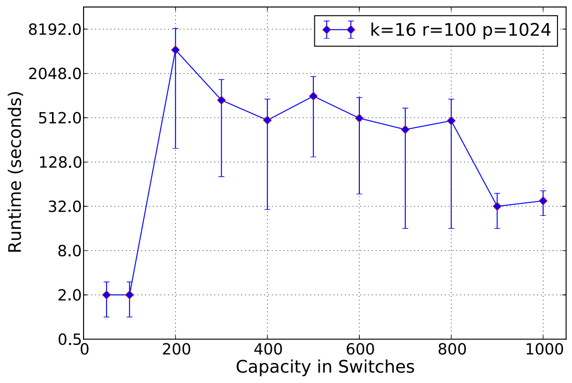


Figure 11. Execution time vs #rule entries per switch.

This assumption may not be valid because routing is based on multiple considerations, and thus may not be permitted to change due to ACL rule placement.

SIMPLE [29] presents an SDN-based policy enforcement layer by steering traffic between different middleboxes. However, SIMPLE can only deal with middlebox level granularity and not rule level as in our approach. Further, it does not optimize rules across multiple paths per policy or across policies. MERLIN [30] proposes a regular expression based policy language and it uses a constraint solver to generate valid policy deployment solutions based on the policy language. Although MERLIN can optimize for bandwidth and link capacities, it does not handle switch rule capacities.

In contrast to these approaches, our solution offers the following advantages:

1) We maximally use rule sharing over multiple routes for a single policy, and over multiples policies.
2) Our approach can handle multiple constraints and optimize many aspects of the network, such as the total number of rules, in a single mathematical optimization framework.
3) Our approach has no false negatives, i.e., we will find a solution if one exists.
4) Our approach is flexible and can be extended to other goals, e.g. routing and load balancing (in the future) and other optimization criteria (e.g. distance from ingress, weighted placement, slack in table capacity, etc.).
5) Our solution is a good match for a multi-tenant datacenter environment. In a datacenter, every tenant has the view that they own their private virtual network but they share the underlying physical network. Therefore, in a datacenter, we have to overlay the policies of multiple virtual networks onto the shared physical network. Every tenant has a disjoint set of virtual ingress ports that map to the same set of physical ingress ports. This can be easily handled in our formulation. It is not clear how other approaches handle virtualization.

Another body of work focuses on the optimization of firewall policies and it is orthogonal to the approach proposed in this paper. For example, [7] studied how to use formal methods (Boolean satisfiability) to remove redundancies in a firewall and it also provided a framework to synthesize an optimal firewall configuration using Quantified Boolean Formulas. Similarly, Liu proposed using decision trees to optimize firewalls in [8], [9]. Other works, such as NetPlumber [24] and VeriFlow [31] ease network management by enabling administrators to check certain network invariants, such as reachability and forwarding loops. However, these methods mainly focus on functional checking and they lack the capability of optimization for rule placement.

Also orthogonal, are other efforts that focus on easier network management through language design. The Frenetic project [32] proposed a declarative network language to specify high-level primitives for the network. Pyretic [33] uses the parallel and sequential composition operators to enable modular programming. Maple [34] provides a centralized algorithmic policy for network programmers.

VII. Concluding Remarks

In this paper we formulate the ACL rule placement problem as ILP and satisfiability problems. We identify the various rule placement constraints, such as switch rule capacity, and show suitable encodings for each case. These formulations capture the sharing of rules across different paths for a given policy, and across different policies. Further, we show how the ILP formulation can be used to optimize different objective functions such as the total rule size.

We also study various aspects of the applicability of this technique to practical sized networks through an experimental evaluation. In particular, we consider scalability in the face of increasing number of rules, paths and different capacity constraints. Overall, in practice this technique is applicable to real-sized networks, e.g. it was successful in placing the rules for a network with 320 switches and a total of 70k rules in 25 seconds of computation time.

Future work includes an experimental evaluation of the Satisfiability formulation using SMT and Pseudo-Boolean solvers. In addition, we plan to explore more complex rule placement constraints, e.g. if the network wants to monitor certain packets, we do not want to let firewall rules block the packets before they reach the monitoring rules.

Acknowledgements

The authors wish to thank the anonymous reviewers and Lujo Bauer for their helpful feedback.

References

[1] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," *Proceedings of ACM CoNEXT*, 2013.

[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.*, Mar. 2008.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, Jul. 2008.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Not.*, Sep. 2011.

[5] *Project Floodlight*, Dec. 2013. [Online]. Available: http://www.projectfloodlight.org

[6] *POX*, Dec. 2013. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[7] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, "Verification and synthesis of firewalls using sat and qbf," in *The 2nd International Workshop on Rigorous Protocol Engineering*, Oct. 2012.

[8] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *Proceedings of the 27th Annual IEEE Conference on Computer Communications (Infocom)*, April 2008.

[9] A. Liu, E. Torng, and C. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, april 2008.

[10] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, Apr. 2012.

[11] C. Meiners, A. Liu, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," in *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, 2007.

[12] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07, 2007.

[13] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *SIGCOMM Comput. Commun. Rev.*, Aug. 2010.

[14] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "vcrib: Virtualized rule management in the cloud," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12, 2012.

[15] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," *IEEE Infocomm Mini-conference*, 2013.

[16] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, Sep. 2011.

[17] E. Boros and P. L. Hammer, "Pseudo-boolean optimization," *Discrete applied mathematics*, 2002.

[18] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," OpenFlow, Tech. Rep., 2009. [Online]. Available: http://archive.openflow.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf

[19] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, Aug. 2011.

[20] *Google Compute Engine*, Apr. 2014. [Online]. Available: https://developers.google.com/compute/docs/networking#firewalls

[21] *Amazon EC2 Security Group*, Apr. 2014. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html

[22] *iptables*, 1998. [Online]. Available: http://www.netfilter.org/projects/iptables/index.html

[23] *Open vSwitch*, 2011. [Online]. Available: http://openvswitch.org

[24] 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.

[25] *CPLEX*, Dec. 2013. [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

[26] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, Aug. 2008.

[27] D. Taylor and J. Turner, "Classbench: a packet classification benchmark," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, march 2005.

[28] P. Gupta and N. McKeown, "Packet classification on multiple fields," *SIGCOMM Comput. Commun. Rev.*, Aug. 1999.

[29] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *SIGCOMM '2013*, 2013.

[30] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with merlin," in *HotNets '2013*, ser. HotNets-XII, 2013.

[31] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI'13, 2013.

[32] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Not.*, Sep. 2011.

[33] 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.

[34] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," *SIGCOMM Comput. Commun. Rev.*, Aug. 2013.