Optimizing the “One Big Switch” Abstraction in Software-Defined Networks

Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker
Computer Science Department, Princeton University

ABSTRACT

Software Defined Networks (SDNs) support diverse network policies by offering direct, network-wide control over how switches handle traffic. Unfortunately, many controller platforms force applications to grapple simultaneously with end-to-end connectivity constraints, routing policy, switch memory limits, and the hop-by-hop interactions between forwarding rules. We believe solutions to this complex problem should be factored in to three distinct parts: (1) high-level SDN applications should define their end-point connectivity policy on top of a “one big switch” abstraction; (2) a mid-level SDN infrastructure layer should decide on the hop-by-hop routing policy; and (3) a compiler should synthesize an effective set of forwarding rules that obey the user-defined policies and adhere to the resource constraints of the underlying hardware. In this paper, we define and implement our proposed architecture, present efficient rule-placement algorithms that distribute forwarding policies across general SDN networks while managing rule-space constraints, and show how to support dynamic, incremental update of policies. We evaluate the effectiveness of our algorithms analytically by providing complexity bounds on their running time and rule space, as well as empirically, using both synthetic benchmarks, and real-world firewall and routing policies.

1. INTRODUCTION

Software-Defined Networking (SDN) enables flexible network policies by allowing controller applications to install packet-handling rules across a distributed collection of switches. Over the past few years, many applications (*e.g.,* server load balancing, virtual-machine migration, and access control) have been built using the popular OpenFlow API [1]. However, many controller platforms [2, 3, 4, 5, 6] force applications to manage the network at the level of individual switches by representing a high-level policy in terms of the rules installed in each switch. This forces programmers to reason about many low-level details, all at the same time, including the choice of path, the rule-space limits on each switch, and the hop-by-hop interaction of rules for forwarding, dropping, modifying, and monitoring packets.

Rule space, in particular, is a scarce commodity on current SDN hardware. Many applications require rules that match on multiple header fields, with “wildcards” for some bits. For example, access-control policies match on the “five tuple” of source and destination IP addresses and port numbers and the protocol [7], whereas a load balancer may match on source and destination IP prefixes [8]. These rules are naturally supported using Ternary Content Addressable Memory (TCAM), which can read all rules in parallel to identify the matching entries for each packet. However, TCAM is expensive and power hungry. The merchant-silicon chipsets in commodity switches typically support just a few thousand or tens of thousands of entries [9].

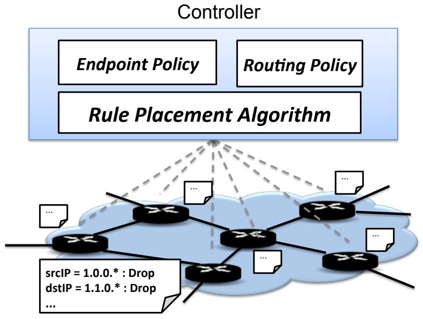


Figure 1: High-level policy and low-level rule placement

Rather than grappling with TCAM sizes, we argue that SDN application programmers should define high-level policies and have the controller platform manage the placement of rules on switches. We, and more broadly the community at large [10, 11, 12], have observed that such high-level policies may be specified in two parts, as shown in Figure 1:

- **An endpoint policy:** Endpoint policies, like access control and load balancing, view the network as *one big switch* that hides internal topology details. The policy specifies which packets to drop, or to forward to specific egress ports, as well as any modifications of header fields.
- **A routing policy:** The routing policy specifies what paths traffic should follow between the ingress and egress ports. The routing policy is driven by traffic-engineering goals, such as minimizing congestion and end-to-end latency.

Expressing these two parts of the policy separately modularizes the problem and allows, for example, an SDN client to express an endpoint policy at the highest level of abstraction while an SDN provider plans the lower-level routing policy. Given two such specifications, the controller platform (a compiler) can apply a *rule-placement algorithm* to generate switch-level rules that realize both parts of the policy correctly, while adhering to switch table-size constraints.

Optimizing the placement of rules is challenging. Minimizing the number of rules for a *single* switch is computationally difficult [13], though effective heuristics exist [14]. Solving the rule-placement problem for an entire *network* of switches, given independent endpoint and routing policies, is even harder:

- Given an optimal rule list for the traffic between two endpoints, we must generate an efficient and correct layout of rules along the path between them.
- The layout must carefully manage the interactions between packet modification and packet forwarding (which may depend upon the fields modified).
- A network consists of multiple paths that share switch rule space. Consequently, we must consider the joint optimization problem across all paths.
- Since network policies evolve dynamically, we must be able to process changes efficiently, without recomputing rule placement from scratch.
- We must manage the fact that in each of the previous tasks, forwarding depends upon packet analysis over multiple dimensions of header fields.

In this paper, we take on the general challenge of solving the rule-placement problem. In doing so, we make a number of important contributions that range from new algorithm design, to complexity analysis, to implementation and empirical analysis on both synthetic and real-world data sets. More specifically, our central contributions include:

- The design of a novel rule-placement algorithm. The algorithm has as a key building block an elegant and provably efficient new technique for rule layout along a linear series of switches.
- The design and analysis of *principled heuristics* for controlling the complexity of our algorithm. These heuristics bring, among other things, the concept of *cost-effective covers* from the broader algorithms literature to bear on rule placement.
- The design of new *algorithms for incremental rule update* when either the endpoint or routing policy change. Such algorithms are a crucial practical component of any SDN system that requires rapid response to a changing environment.
- An *evaluation* of our algorithms in terms of rule space and running time on both synthetic and real-world data that validates our algorithm.

In the next section, we formally introduce the optimization problem required to implement the *one big switch* abstraction. Next, Section 3 presents related work on optimizing rule space. Section 4 presents our algorithm and Section 5 addresses the incremental update issues. Section 6 presents experiments involving both synthetic benchmarks and real-world policies. We conclude the paper in Section 7. A technical report [15] presents the missing proofs and additional experiments.

2. OPTIMIZING A BIG SWITCH

A key problem in implementing the *one big switch* abstraction is mapping global, high-level policies to an equivalent, low-level set of rules for each switch in the network. We call this problem the *big switch problem*, and introduce a precise formulation in this section.

Network topology: The network consists of n switches, each with a set of ports. We refer a port at a switch as a *location* (loc). The locations connected to the outside world are *exposed locations*. A packet enters the network from an exposed location called *ingress* and leaves at an exposed location called *egress*.

Packets: A packet (pkt) includes multiple header fields. Examples of header fields include source IP (src_ip) and destination IP (dst_ip). Switches decide how to handle traffic based on the header fields, and do not modify any other part of the packet; hence, we equate a packet with its header fields.

Switches: Each switch has a single, prioritized list of rules $[r_1, \ldots, r_k]$, where rule r_i has a predicate $r_i.p$ and an action $r_i.a$. A *predicate* is a boolean function that maps a packet header and a location (pkt, loc) into $\{\text{true}, \text{false}\}$. A predicate can be represented as a conjunction of clauses, each of which does *prefix* or *exact* matching on a single field or location. An action could be either "drop" or "modify and forward". The "modify and forward" action specifies how the packet is modified (if at all) and where the packet is forwarded. Upon receiving a packet, the switch identifies the *highest-priority* rule with a matching predicate, and performs the associated action. A packet that matches no rules is dropped by default.

Endpoint policy (E): The endpoint policy operates over the set of exposed locations as if they were ports on one big abstract switch. An endpoint policy is a prioritized list of rules $E \triangleq [r_1, ..., r_m]$, where $m = \|E\|$ is the number of rules. We assume that at the time a given policy E is in effect, each packet can enter the network through at most one ingress point (*e.g.,* the port connected to the sending host, or an Internet gateway).

Routing policy (R): A routing policy R is a function $R(\text{loc}_1, \text{loc}_2, \text{pkt}) = s_{i_1} s_{i_2} ... s_{i_k}$, where loc_1 denotes packet ingress, loc_2 denotes packet egress. The sequence $s_{i_1} s_{i_2} ... s_{i_k}$ is the path through the network. The routing policy may direct all traffic from loc_1 to loc_2 over the same path, or split the traffic over multiple paths based on packet-header fields.

Rule-placement problem: The inputs to the rule-placement problem are the network topology, the endpoint policy E, the routing policy R, and the maximum number of rules each physical switch can hold. The output is a list of rules on each switch such that the network (i) obeys the endpoint policy E, (ii) forwards the packets over the paths specified by R, and (iii) does not exceed the rule space on each switch.

3. RELATED WORK

Prior work on rule-space compression falls into four main categories, as summarized in Table 1.

Compressing policy on a single switch: These algorithms reduce the number of rules needed to realize a policy on *a single switch*. While orthogonal to our work, we can leverage these techniques to (i) reduce the size of the endpoint policy that is input to our rule-placement algorithm and (ii) further optimize the per-switch rule-lists output by our algorithm.

Distributing policy over the network perimeter: These works distribute a centralized firewall policy by placing rules for packets at their ingress switches [7, 17], or verify that the edge switch configurations realize the firewall policy [18]. These algorithms do not enforce rule-table constraints on the edge switches, or place rules on the internal switches; thus, we cannot directly adopt them to solve our problem.

Distributing policy while changing routing: DIFANE [19] and vCRIB [20] leverage all switches in the network to enforce an endpoint policy. They both direct traffic through intermediate switches that enforce portions of the policy, deviating from the routing policy given by users. DIFANE takes a "rule split and caching" approach that increases the path length for the first packet of a flow, whereas vCRIB directs all packets of some flows over longer paths. Instead, we view routing policy as something the SDN *application* should control, based on higher-level goals like traffic engineering. As such, our algorithms must grapple with optimizing rule placement while respecting the routing policy.

Distributing policy while respecting the routing: Similar to our solution, Palette [21] takes both an endpoint policy and routing policy as input, and outputs a rule placement. However, Palette leads to suboptimal solutions for two main reasons. First, Palette has *all* network paths fully implement the endpoint policy. Instead, we only enforce the portion of the endpoint policy affecting the packets on each path. Second, the performance of their algorithm depends on the length of the shortest path (with non-zero traffic) in the network. The algorithm cannot use all available switches when the shortest path's length is small, as is the case for many real networks. Section 6 experimentally compares Palette with our algorithm. We remark here that existing packet-classification algorithms [22, 23] could be viewed as a special case of Palette's partitioning algorithm. These techniques are related to a module in our algorithm, but they cannot directly solve the rule-placement problem. We make further comparisons with these techniques when we present our algorithm.

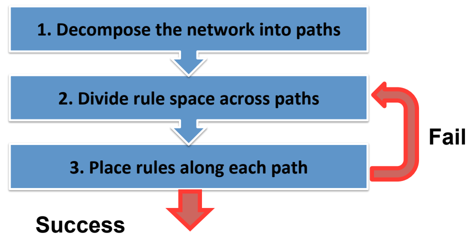

1. Decompose the network into paths
2. Divide rule space across paths
3. Place rules along each path
Fail
Success

Figure 2: Overview of the rule placement algorithm

4. A "ONE BIG SWITCH" ALGORITHM

This section describes our algorithm, which leverages the following two important observations:

The "path problem" is a building block: Given a packet pkt entering the network, the function R gives the route $s_1, ..., s_\ell$ for the packet. We must ensure that E is correctly applied to pkt along the path. Therefore, deciding the rule placement for the path $s_1, ..., s_\ell$ to implement E is a basic building block in our algorithm.

The "path problem" is an easier special case of our problem: We may also interpret the path problem as a special case of our general problem, where the network topology degenerates to a path. Thus, algorithmically understanding this special case is an important step towards tackling the general problem.

The high-level idea of our algorithm is to find an effective way to decompose the general problem into smaller problems over paths and design efficient heuristics to solve the path problems.

4.1 Algorithm Overview

Figure 2 shows the three main components of our algorithm:

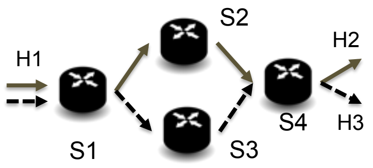
Decomposition (component 1): Our first step is to interpret the problem of implementing $\{E, R\}$ as implementing the routing policy R and a *union* of endpoint policies over the *paths*. We give an example for the routing policy in Figure 3(a) and the endpoint policy in Figure 3(b). From these policies, we can infer that the packets can be partitioned into two groups (see Figure 3(c)): the ones in D_1 (using the path $P_1 = s_1s_2s_4$) and the ones in D_2 (using the path $P_2 = s_1s_3s_4$). We may separately implement two path-wise endpoint policies on P_1 and P_2. By doing so, we decompose the general rule-placement problem into smaller sub-problems. More formally, we can associate path P_i with a *flow*

Type and Examples	Switches	Rule-space limits	Routing policy
Compressing policy on a single switch [14, 16, 13]	Single	Yes	N/A
Distributed policies on edge [7, 17, 18]	Edge	No	Yes
Distributed policies while changing routing [19, 20]	All	Yes	No
Distributed policies while respecting routing [21]	Most	Yes	Yes
Our work	All	Yes	Yes

Table 1: Summary of related works

$$r_1 : (\text{dst\_ip} = 00*, \text{ingress} = H_1 : \text{Permit}, \text{egress} = H_2)$$
$$r_2 : (\text{dst\_ip} = 01*, \text{ingress} = H_1 : \text{Permit}, \text{egress} = H_3)$$

(a) An example endpoint policy E



(b) An example routing policy R

$$P_1 = s_1 s_2 s_4, D_1 = \{\text{dst\_ip} = 00*\}$$
$$P_2 = s_1 s_3 s_4, D_2 = \{\text{dst\_ip} = 01*\}$$

(c) Paths and flow spaces computed from E and R

Figure 3: An example decomposition

space D_i (*i.e.,* all packets that use P_i belong to D_i)[1] and the *projection* E_i of the endpoint policy on D_i:

$$E_i(\text{pkt}) = \begin{cases} E(\text{pkt}) & \text{if pkt} \in D_i \\ \perp & \text{otherwise,} \end{cases} \quad (1)$$

where $\perp$ means no operation is performed.

Resource allocation (component 2): The per-path problems are not independent, since one switch could be shared by multiple paths (*e.g.,* s_4 in Figure 3). Thus, we must divide the rule space in each switch across the paths, so that each path has enough space to implement its part of the endpoint policy. Our algorithm *estimates* the resources needed for each path, based on analysis of the policy's structure. Then the algorithm translates the resource demands into linear programming (LP) instances and invokes a standard LP-solver. At the end of this step, each path-wise endpoint policy knows how many rules it can use at each switch along its path.

Path algorithm (component 3): Given the rule-space allocation for each path, the last component generates a rule placement for each path-wise policy. For each path, the algorithm efficiently searches for an efficient "cover" of a portion of the rules and "packs" them into the switch, before moving on to the next switch in the path. If the estimate of the rule-space requirements in step 2 was not accurate, the path algorithm may fail to find a feasible rule placement, requiring us to repeat the second and third steps in Figure 2.

[1]We can associate a dropped packet with the most natural path it belongs to (*e.g.,*the path taken by other packets with the same destination address).

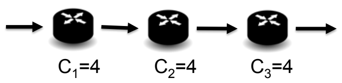


Figure 4: An 3-hop path

The rest of this section presents these three components from the bottom up. We start with the path algorithm (component 3), followed by the solution for general network topologies (components 1 and 2). We also discuss extensions to the algorithm to enforce endpoint policies as early in a path as possible, to minimize the overhead of carrying unwanted traffic.

4.2 Placing Rules Along a Path

Along a path, every switch allocates a fixed rule capacity to the path, as in Figure 4. For a single path P_i, the routing policy is simple—all packets in the flow space D_i are forwarded along the path. We can enforce this policy by installing fixed forwarding rules on each switch to pass packets to the next hop. The endpoint policy is more complex, specifying different actions for different packets in the flow space. Along the path, we need to perform the action (such as drop or modification) for each packet *exactly once*. However, *where* the packet is processed (or where the rules are placed), is constrained by the rule capacity of the switches. Therefore, the endpoint policy is the one that gives us flexibility to moves rules among multiple switches.

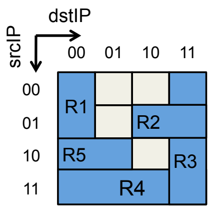
Our goal is to minimize the number of rules needed to realize the endpoint policy,[2] while respecting the rule-capacity constraints.[3] In what follows, we present a heuristic that recursively *covers* the rules and *packs* groups of rules into switches along the path. This algorithm is computationally efficient and offers good performance. For ease of exposition, we assume the flow space associated with the path is the full space (containing all packets) and the endpoint policy matches on the source and destination IP prefixes (two-dimensional). A fully general version of the algorithm is presented at

[2]Since optimizing the rule list for a single switch is NP-hard [24], we cannot assume an optimal representation of E is provided as input to our algorithm. Instead, we accept *any* prioritized list of two-dimensional rules. In practice, the application module generating the endpoint policy may optimize the representation of E.

[3]There exists a standard reduction between decision problems and optimization problems [25]. So we will switch between these two formulations whenever needed.

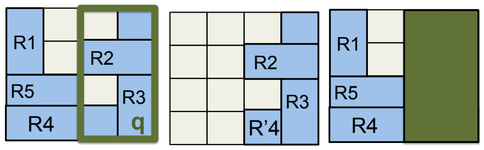
$R1$: (src_ip = 0*, dst_ip = 00 : Permit)
$R2$: (src_ip = 01, dst_ip = 1* : Permit)
$R3$: (src_ip = *, dst_ip = 11 : Drop)
$R4$: (src_ip = 11, dst_ip = * : Permit)
$R5$: (src_ip = 10, dst_ip = 0* : Permit)
$R6$: (src_ip = *, dst_ip = * : Drop)

(a) Prioritized rule list of an access-control policy



(b) Rectangular representation of the policy

Figure 5: An example two-dimensional policy



(a) Cover q (b) Switch s_1 (c) After s_1

Figure 6: Processing 2-dim endpoint policy E

the end of this subsection.

4.2.1 Overview of path heuristic

The endpoint policy E can be viewed as a two-dimensional space where each rule is mapped to a rectangle based on the predicate, and higher-priority rectangles lie on top of lower-priority rectangles, as shown in Figure 5.[4] Let us consider enforcing the policy on the path shown in Figure 4. Since a single switch cannot store all six rules from Figure 5(a), we must divide the rules across multiple switches. Our algorithm recursively *covers* a rectangle, *packs* the overlapping rules into a switch, and *replaces* the rectangle region with a single rule, as shown in Figure 6.

Cover: The "cover" phase selects a rectangle q as shown in Figure 6(a). The rectangle q overlaps with rules $R2$, $R3$, $R4$, and $R6$ (overlapping rules), with $R2$ and $R3$ (internal rules) lying completely inside q. We require that the number of overlapping rules of a rectangle *does not exceed* the rule capacity of a single switch.

Pack: The intersection of rectangle q and the overlapping rules (see Figure 6(b)) defines actions for packets inside the rectangle. The intersection can also be viewed as the projection of the endpoint policy E on q, denoted as E_q (Figure 7(a)).
By "packing" E_q on the current switch, all packets falling into q are processed (*e.g.,* dropped or permitted), and the remaining packets are forwarded to the next switch.

[4]Rule $R6$ is intentionally omitted in the figure since it covers the whole rectangle.

(a) E_q
$r_1 : (q \wedge R2.p, R2.a)$
$r_2 : (q \wedge R3.p, R3.a)$
$r_3 : (q \wedge R4.p, R4.a)$
$r_4 : (q \wedge R6.p, R6.a)$

(b) New rule list
$r_1 : (q, \text{Fwd})$
$r_2 : (R1.p, R1.a)$
$r_3 : (R4.p, R4.a)$
$r_4 : (R5.p, R5.a)$
$r_6 : (R6.p, R6.a)$

Figure 7: Example policy

Replace: After packing the projection E_q in a switch, we *rewrite* the endpoint policy to avoid re-processing the packets in q: we first add a rule $q^{\text{Fwd}} = (q, \text{Fwd})$ with the highest priority to the policy. The rule q^{Fwd} forwards all the packets falling in q without any modification. Second, all internal rules inside q ($R2$ and $R3$) can be safely deleted because no packets will ever match them. The new rewritten endpoint policy and corresponding rule list are shown in Figure 6(c) and Figure 7(b).

The cover-pack-and-replace operation is recursively applied to distribute the rewritten endpoint policy over the rest of the path. Our heuristic is "greedy": at each switch, we repeatedly pack rules as long as there is rule space available before proceeding to the next switch. We make two observations about the cover-pack-and-replace operation:

- *Whether a feasible rule placement exists becomes clear upon reaching the last switch in the path.* If we can fit all remaining rules on the last switch, then the policy can be successfully implemented; otherwise, no feasible rule placement exists.
- *The total number of installed rules will be no less than the number of rules in the endpoint policy.* This is primarily because only rules inside the rectangle are deleted. A rule that partially overlaps with the selected rectangle will appear on multiple switches. Secondly, additional rules (q, Fwd) are included for every rectangle to avoid re-processing.

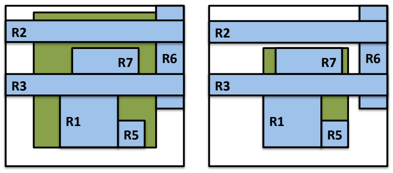
4.2.2 Search for candidate rectangles

Building on the basic framework, we explore *what* rectangle to select and *how* to find the rectangle.

Rectangle selection plays a significant role in determining the efficacy of rule placement. A seemingly natural approach is to find a predicate q that completely covers as many rules as possible, allowing us to remove the most rules from the endpoint policy. However, we must also consider the cost of duplicating the partially-overlapping rules. Imagine we have two candidate rectangles q_1 (with 10 internal rules and 30 overlapping rules) and q_2 (with 5 internal rules and 8 overlapping rules). While q_1 would allow us to delete more rules, q_2 makes more effective use of the rule space. Indeed, we can define the cost-effectiveness of q in a natural way:

$$utility(q) = \frac{\#\text{internal rules} - 1}{\#\text{overlapping rules}}$$

If q is selected, all overlapping rules must be installed on



(a) Large cover (b) Small cover

Figure 8: Not using unnecessarily large cover.

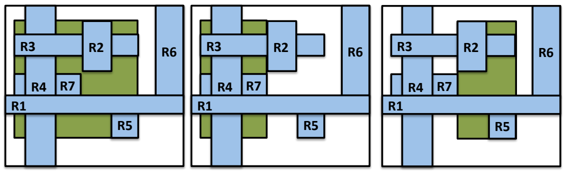
the switch, while only the internal rules can be removed and one extra rule (for q^{Fwd}) must be added[5].

Top-Down search strategy is used in finding the most cost-effective rectangle. We start with rectangle (src_ip $=$ $*$, dst_ip $= *$), and expand the subrectangles (src_ip $=$ $0*$, dst_ip $= *$), (src_ip $= 1*$, dst_ip $= *$), (src_ip $=$ $*$, dst_ip $= 0*$), and (src_ip $= *$, dst_ip $= 1*$). In the search procedure, we always shrink the rectangle to align with rules, as illustrated by the example in Figure 8. Suppose our algorithm selected the predicate p in Figure 8(a) (the shadowed one) to cover the rules. We can shrink the predicate as much as possible, as long as the set of rules fully covered by p remains unchanged. Specifically, we may shrink p as illustrated in Figure 8(b), without impacting the correctness of the algorithm. Moroever, for any shrinked predicate, two rules determine the left and right boundaries on the x-axis, resulting in a total of m^2 possible sides along the x-axis. Similarly, the y-axis has a total of m^2 possible sides, resulting in a total number of relevant predicates of m^4.

Even searching $O(m^4)$ predicates in each pack-cover-and-replace operation would be impractical for larger m. To limit the search space, our algorithm avoids searching too deeply, preferring larger rectangles over smaller ones. Specifically, let q be a rectangle and q' be its subrectangle (q' is inside q). When both Q_q and $Q_{q'}$ can "fit" into the same switch, packing Q_q often helps reduce the number of repeated rules. In Figure 9, we can use either the large rectangle in Figure 9(a) or the two smaller rectangles in Figure 9(b). Using the larger rectangle allows us to remove $R3$. Using the two smaller rectangles forces us to repeat $R3$, and repeat $R1$ and $R4$ one extra time. As such, our algorithm avoids exploring all of the small rectangles. Formally, we only consider those rectangles q such that there *exists no* rectangle q' which satisfies both of the following two conditions: (i) q is inside q' and (ii) $Q_{q'}$ can be packed in the switch. We call these q the *maximal feasible* predicates.

The pseudocode of the full path heuristic is shown in Figure 10. Note that we could have used an existing rule-space partitioning algorithm [22, 23, 27, 28] but they are less effective. The works of [22, 23, 27] take a top-down approach to recursively cut a cover into smaller ones until each of the cover fits into one switch. This approach cannot ensure that every switch fully uses its space[6] (See results in Section 6). SmartPC [28] takes a bottom-up approach to find the covers. But it searches much less extensively among the set of feasible covers; they do not use the cost-effectiveness metric to control the number of repeated rules either.

[5]Cost-effectiveness metrics have been used in other domains to solve covering problems [26].



(a) using larger p (b) using smaller rectangles

Figure 9: Only pack maximal rectangle.

```
PACK-AND-REPLACE-2D($i, E'$)
1  Let $d_i \leftarrow$ the remaining capacity of $s_i$.
2  Let $Q$ be the set of maximal feasible predicates
       with respect to $E'$ that need $\leq d_i$ rules.
3  $q \leftarrow \arg\max_q \{\frac{|\text{Internal}(q,E')|-1}{\|E'_q\|} \mid q \in Q\}$
4  Append the rules $E'_q$ sequentially to the end of
       the prioritized list of $s_i$.
5  Let $R \leftarrow \{r \in E' : r.p$ is inside $q\}$.
6  $E' \leftarrow E' \backslash R$
7  $E' \leftarrow (q, \text{Fwd}) \circ E'$

COMPILE-RULES-2D($\{s_1, ..., s_\ell\}, E$)
1  $E' \leftarrow E$
2  for $i \leftarrow 1$ to $\ell$
3      do Add a default forward rule for all
              unmatched packet at $s_i$.
4         while $s_i$ has unused rule space
5             do PACK-AND-REPLACE($i, E'$)
```

Figure 10: Our heuristics for 2-dim chains

4.2.3 *Generalizing the algorithm*

We end this subsection by highlighting a number of extensions to our algorithm.

Smaller flow space : When the flow space for a path is not the full space, we can still use the algorithm except we require that the rectangular covers chosen for the path reside in the corresponding flow space.

Single dimension: If the endpoint policy depends on only one header field, this algorithm is near optimal—the gap between the number of installed rules and the size of endpoint policy is marginally small. See [15] for a discussion.

Higher dimensions: Our algorithm works when Q is a d-dimensional function for a $d \geq 3$—by (i) still cutting the along the two dimensions of source and destination IP prefix with all rules projected to rectangles, or (ii)

[6]For example, imagine at some point in their algorithms, one cover contains $C + 1$ rules. Since this cannot fit into one switch, they cut the cover further into two smaller ones. But after this cut, each of the two subcovers could have only $\approx C/2$ rules, wasting nearly 50% of space in each switch.

using "hypercube" predicates instead of rectangular and cutting on all dimensions.

Switch-order independence: Once our algorithm finishes installing rules on switches, we are able to swap the contents of any two switches without altering the behavior of the whole path (the full technical report gives a proof for this claim). This property plays a key role when we tackle the general graph problem.

4.3 Decomposition and Allocation

We now describe how we decompose the network problem into paths and divide rule space over the paths.

4.3.1 Decomposition through cross-product

We start the "decomposition" by finding all τ paths in the graph $P_1, P_2, ..., P_\tau$, where path P_i is a chain of switches $s_{i_1} s_{i_2} ... s_{i_{\ell_i}}$ connecting one exposed location to another. By examining the "cross-product" of endpoint policy E and routing policy R, we find the flow space D_i for each P_i. Finally, we project the endpoint policy E on D_i to compute the "sub-policy" E_i for P_i. This generates a collection of path problems: for each path P_i, the endpoint policy is E_i and the routing policy directs all packets in D_i over path P_i.

Since each packet can enter the network via a single ingress location and exit at most one egress location (Section 2), any two flow spaces D_i and D_j are *disjoint*. Therefore, we can solve the rule-placement problem for each path separately. In addition to solving the τ rule-placement problems, we must ensure that switches correctly forward traffic along all paths, *i.e.,* the routing policy is realized. The algorithm achieves this by placing low-priority *default rules* on switches. These default rules enforce "forward any packets in D_i to the next hop in P_i", such that packets that are not handled by higher-priority rules traverse the desired path.

4.3.2 Rule allocation through linear programming

Ideally, we would simply solve the rule-placement problem separately for each path and combine the results into a complete solution. However, multiple paths can traverse the same switch and need to share the rule space. For each path P_i, we need to allocate enough rule space at each switch in P_i to successfully implement the endpoint policy E_i, while respecting the capacity constraints of the switches. The goal of "allocation" phase is to find a *global* rule-space allocation, such that it is feasible to find rule placements for *all* paths.

Enumerating all possible rule-space partitions (and checking the feasibility by running the path heuristic for each path) would be too computationally expensive. Instead, we capitalize on a key observation from evaluating our path heuristic: the feasibility of a rule-space allocation depends primarily on the *total* amount of space allocated to a path, rather than the *portion* of that space allocated to each switch. That is, if the path heuristic can find a feasible rule placement for Figure 4 under the allocation $(c_1 = 4, c_2 = 4, c_3 = 4)$, then the heuristic is likely to work for the allocation $(c_1 = 3, c_2 = 4, c_3 = 5)$, since both allocations have space for 12 rules.

$$\begin{array}{lll} \text{max:} & & \perp \\ \text{s.t:} & \forall i \le n: & \sum_{j \le \tau} h_{i,j} \cdot x_{i,j} \le 1 \quad \text{(C1)} \\ & \forall j \le \tau: & \sum_{i \le n} h_{i,j} \cdot x_{i,j} \cdot c_j \ge m_j \quad \text{(C2)} \end{array}$$

Figure 11: Linear program for rule-space allocation

To assess the feasibility of a rule-space allocation plan, we introduce a threshold value η for the given path: if the total rule space allocated by the plan is no less than η ($c_1 + c_2 + c_3 \ge \eta$ in the example), then a feasible rule placement is likely to exist; otherwise, there is no feasible rule placement. Therefore, our rule-space allocation plan consists of two steps: (i) estimate the threshold value η for each path and (ii) compute a global rule-space allocation plan, which satisfies all of the constraints on the threshold values.

This strategy is very efficient and avoids exhaustive enumeration of allocation plans. Furthermore, it allows us to estimate whether *any* feasible solution exists without running the path heuristics.

Estimate the necessary rule space per path: Two factors impact the total rule space needed by a path:

- *The size of endpoint policy:* The more rules in the endpoint policy, the more rule space is needed.
- *The path length:* The number of rectangles grows with the length of the path, since each switch uses at least one rectangle.

Since paths have different endpoint policies and lengths, we estimate the threshold value for the ℓ_i-hop path P_i with endpoint policy E_i. When $\sum_{j \le \ell_i} c_{i_j} \ge \eta_i$ for a suitably chosen η_i, a feasible solution is likely to exist. In practice, we found that η_i grows linearly with $\|E_i\|$ and ℓ_i. Thus, we set $\eta_i = \alpha_i \|E_i\|$, where α_i is linear in the length of P_i and can be estimated empirically.

Compute the rule-space allocation: Given the space estimates, we partition the capacity of each switch to satisfy the needs of all paths. The decision can be formulated as a linear programming problem (hereafter LP). Switch s_i can store c_i rules, beyond the rules needed for the default routing for each path. Let m_j be the estimated total rule space needed by path P_j. We define $\{h_{i,j}\}_{i \le n, j \le \tau}$ as indicator variables so that $h_{i,j} = 1$ if and only if s_i is on the path P_j. The variables are $\{x_{i,j}\}_{i \le n, j \le \tau}$, where $x_{i,j}$ represents the portion of rules at s_i allocated to P_j. For example, when $c_4 = 1000$ and $x_{4,3} = 0.4$, we need to allocate $1000 \times 0.4 = 400$ rules at s_4 for the path P_3. The LP has two types of constraints (see Figure 11): (i) capacity constraints ensuring that each switch s_i allocates no more than 100%

of its available space and (ii) path constraints ensuring that each path P_j has a total space of at least m_j.

Our LP does not have an objective function since we are happy with any assignment that satisfies all the constraints.[7] Moreover, we apply floor functions to round down the fractional variables, so we never violate capacity constraints; this causes each path can lose at most ℓ_i rules compared to the optimal solution, where the path length ℓ_i is negligibly small.

Re-execution and correctness of the algorithm: When the path algorithm fails to find a feasible solution based on the resource allocation computed by our LP, it means our threshold estimates are not accurate enough. In this case, we increase the thresholds for the failed paths and re-execute the LP and path algorithms and repeat until we find a feasible solution. In the technical report, we show the correctness of the algorithm.

4.4 Minimizing Unwanted Traffic

One inevitable cost of distributing the endpoint policy is that some unwanted packets travel one or more hops before they are ultimately dropped. For instance, consider an access-control policy implemented on a chain. Installing the entire endpoint policy at the ingress switch would ensure all packets are dropped at the earliest possible moment. However, this solution does not utilize the rule space in downstream switches. In its current form, our algorithm distributes rules over the switches without regard to where the traffic gets dropped. A simple extension to our algorithm can minimize the cost of carrying unwanted packets in the network. Specifically, we leverage the following two techniques:

Change the LP's objective to prefer space at the ingress switches: In our original linear program formulation, we do not set any objective. When we execute a standard solver on our LP instance, we could get a solution that fully uses the space in the "interior switches," while leaving unused space at the edge. This problem becomes more pronounced when the network has more rule space than the policy needs (*i.e.,* many feasible solutions exist). When our path algorithm runs over space allocations that mostly stress the interior switches, the resulting rule placement would process most packets deep inside the network. We address this problem by introducing an objective function in the LP that prefers a solution that uses space at or near the first switch on a path. Specifically, let ℓ_j be the length of the path P_j. Our objective is

$$\max: \sum_{i \leq n} \sum_{j \leq \tau} \frac{\ell_j - w_{i,j} + 1}{\ell_j} h_{i,j} x_{i,j}, \qquad (2)$$

where $w_{i,j}$ is s_i's position in P_j. For example, if s_4 is the third hop in P_6, then $w_{4,6} = 3$.

[7]This is still equivalent to standard linear programs; see [29].

Leverage the switch-order independence in the path algorithm: At the path level, we can also leverage the switch-order independence property discussed in Section 4.2.3 to further reduce unwanted traffic. Specifically, notice that in our path algorithm, we sequentially pack and replace the endpoint policies over the switches. Thus, in this strategy, more fine-grained rules are processed first and the "biggest" rule (covering the largest amount of flow space) is processed at the end. On the other hand, the biggest rule is more likely to cover larger volumes of unwanted traffic. Thus, putting the biggest rules at or near the ingress will drop unwanted traffic earlier. This motivates us to *reverse the order* we place rules along a chain: here, we shall first pack the most refined rules at the last switch, and progressively pack the rules in upstream switches, making the ingress switch responsible for the biggest rules.

5. INCREMENTAL UPDATES

Network policies change over time. Rather than computing a new rule placement from scratch, we must update the policy *incrementally* to minimize the computation time and network disruption. We focus on the following major practical scenarios for policy updates:

Change of drop or modification actions: The endpoint policy may change the subset of packets that are dropped or how they are modified. A typical example is updating an access-control list. Here, the flow space associated with each path does not change.

Change of egress points: The endpoint policy may change where some packets leave the network (*e.g.,* because a mobile destination moves). Here, the flow space changes, but the routing policy remains the same.

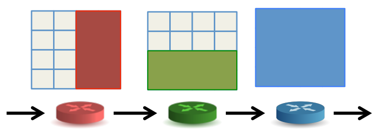
Change of routing policy: When the topology changes, the routing policy also need to be changed. In this case, the network has some new paths, and the flow space may change for some existing paths.

The first example is a "planned change," while the other two examples may be planned (*e.g.,* virtual-machine migration or network maintenance) or unplanned (*e.g.,* user mobility or link failure). While we must react quickly to unplanned changes to prevent disruptions, we can handle planned updates more slowly if needed. These observations guide our algorithm design, which has two main components: a "local algorithm" used when the flow space does not change, and a "global algorithm" used when the flow space does change. [8]

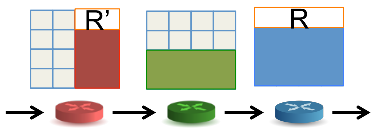
5.1 Local Algorithm

When the flowspace remains the same (*i.e.,* all packets continue to traverse the same paths), a local update algorithm is sufficient. If a path's policy does not change, the rule placement for that path does not need

[8]We can use techniques in [30] to ensure consistent updates.



(a) Original placement



(b) Final placement

Figure 12: Rule insertion example

```
INSERT-RULE-PATH($\{s_1, ..., s_\ell\}, R, E$)
1  for $i \leftarrow 1$ to $\ell$
2        do Let $Q$ be the set of predicates covered by $s_i$.
3           for every predicate $q \in Q$
4               do if $R.p$ overlaps with $q$
5                     then Install $(R.p \wedge q, R.a)$ on $s_i$
6               do if $R.p$ is inside $q$
7                     then return
```

Figure 13: Procedure for rule insertion

to change, so we *do not* need to re-execute the path algorithm presented in Section 4.2. We can always convert an original path-wise endpoint policy into the new one by applying one of the following three operations one or more times: (i) insert a new rule, (ii) delete an existing rule, and (iii) alter an existing rule. Thus, we need only design an algorithm to handle each of these operations. Then we may recursively invoke this algorithm to update the policy for the entire path.

Let us focus on rule insertion, *i.e.,* adding a new rule R to the endpoint policy E and the path $P = s_1 s_2 ... s_\ell$, as shown in Figure 12. Strategies to handle the other two operations are similar. Recall each switch s_i along the path is responsible for some region of flow space, indicated by predicates. In our algorithm, we simply walk through each s_i and see whether $R.p$ overlaps with the region ($R.p$ is the predicate of rule R). When an overlap exists, we "sneak in" the projection of R with respect to the region of s_i. Otherwise, we do nothing. Figure 13 illustrates the pseudocode.

5.2 Global Algorithm

When the flowspaces change, our algorithm first changes the forwarding rules for the affected paths. Then we must decide the rule placements on these paths to implement the new policies. This consists of two steps. First, we run the linear program discussed in Section 4 *only on the affected paths* to compute the rule-space allocation (notice that rule spaces assigned to unaffected paths should be excluded in the LP). Second, we run the path algorithm for each of the paths using the rule space assigned by the LP.[9]

Performance in unplanned changes. When a switch or link fails, we must execute the global algorithm to find the new rule placement. The global algorithm could be computationally demanding, leading to undesirable delays.[10] To respond more quickly, we can precompute a backup rule placement for possible failures and cache the results at the controller. We leave it as a future work to understand the most efficient way to implement this precompute-and-cache solution.

6. PERFORMANCE EVALUATION

In this section, we use real and synthetic policies to evaluate our algorithm in terms of (i) rule-space overhead, (ii) running time, and (iii) resources consumed by unwanted traffic.

6.1 Experimental Workloads

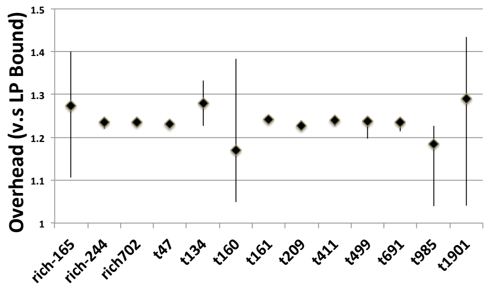
Routing policies: We use GT-ITM [31] to generate 10 synthetic 100-node topologies. Four core switches are connected to each other, and the other 96 switches constitute 12 sub-graphs, each connected to one of the core switches. On average, 53 of these 96 switches lie at the periphery of the network. We compute the shortest paths between all pairs of edge switches. The average path length is 7, and the longest path has 12 hops.

Endpoint policies: We use real firewall configurations from a large university network. There are 13 test cases in total. We take three steps to associate the rule sets with the topology. First, we infer subnet structures using the following observation: when the predicate of a rule list is (src_ip $= q_1 \wedge$ dst_ip $= q_2$) (where q_1 and q_2 are prefixes), then q_1 and q_2 should belong to different subnets. We use this principle to split the IP address space into subnets such that for any rule in the ACL, its src_ip prefix and dst_ip prefix belong to different subnets. Subnets that do not overlap with any rules in the ACL are discarded. Second, we attach subnets to edge switches. Third, for any pair of source and destination subnets, we compute the projection of the ACL on their prefixes. Then we get the final endpoint policy E. 4 of the 13 endpoint policies have less than 15,000 rules, and the rest have 20,000–120,000 rules.

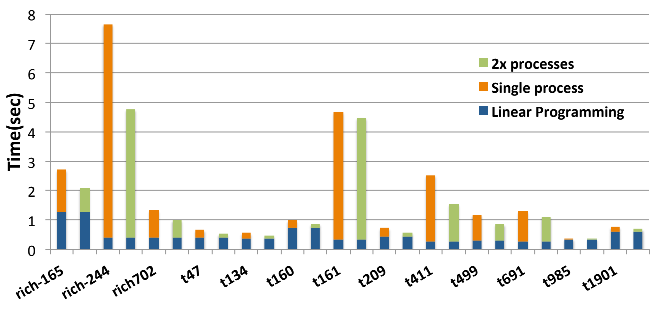
In addition, ClassBench [32] is used to generate synthetic 5-field rule sets to test our path heuristic. ClassBench gives us 12 test cases, covering three typical packet-classification applications: five ACLs, five Firewalls, and two IP Chains. (IP Chain test cases are the decision tree formats for security, VPN, and NAT filter for software-based systems, see [32] for details.)

[9]If the algorithm cannot find an allocation plan leading to feasible rule placements for all affected paths, an overall recomputation must be performed.

[10]In our experiments, we observe the failure of one important switch can cause the recomputation for up to 20% of the paths (see Section 6 for details). The update algorithm may take up to 5 to 10 seconds when this happens.



(a) Rule-space overhead



(b) Computation time (for one and two processes)

Figure 14: The performance of the graph algorithm over different endpoint policies on 100-switch topologies

We evaluate our algorithms using two platforms. For stand-alone path heuristic, we use RX200 S6 servers with dual, six-core 3.06 Intel X5675 processors with 48GB ram. To test the algorithm on graphs, we use the Gurobi Solver to solve linear programs. Unfortunately, the Gurobi Solver is not supported on the RX200 S6 servers so we use a Macbook with OS X 10.8 with a 2.6 GHz Intel Core i7 processor and 8GB memory for the general graph algorithms. Our algorithms are implemented in Java and `C++` respectively.

6.2 Rule-Space Utilization

Our evaluation of rule-space utilization characterizes the *overhead* of the algorithm, defined as the number of extra rules needed to implement the endpoint policy E. The overhead comes from two sources:

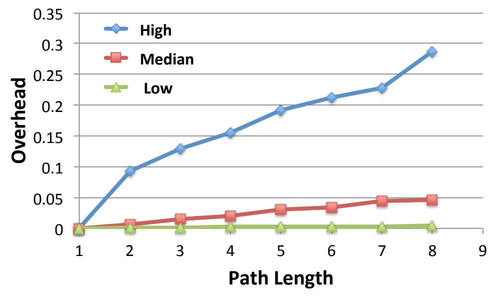
Decomposition of graph into paths: A single rule in the endpoint policy may need to appear on multiple paths. For example, a rule (src_ip = 1.2.3.4 : Drop) matches packets with different destinations that follow different paths; as such, this rule appears in the projected endpoint policies of multiple paths. Our experiments show that this overhead is very small on real policies. The average number of extra rules is typically just twice the number of paths, *e.g.,* in a network with 50 paths and 30k rules in the endpoint policy, the decomposition leads to approximately 100 extra rules.

Distributing rules over a path: Our path heuristic installs additional rules to distribute the path-wide endpoint policy. If our heuristic do a good job in selecting rectangles, the number of extra rules should be small. We mainly focus on understanding this overhead, by comparing to a lower bound of $\|E_i\|$ rules that assumes *no overhead* for deploying rules along path P_i. This also corresponds with finding a solution in our linear program where all α_i's are set to 1.

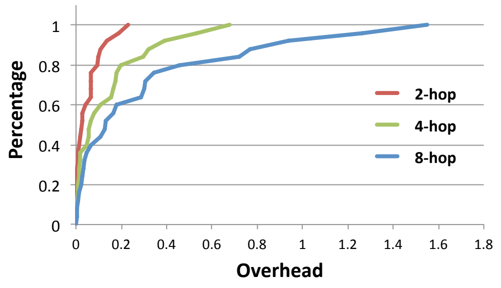
Our experiments assume all switches have the *same* rule capacity. As such, the overhead is defined as $\frac{C-C^L}{C^L}$, where C is the rule capacity of a single switch, such that our algorithm produces a feasible rule placement, and C^L is the minimum rule capacity given by the LP, assuming no overhead is incurred in the path algorithm.

Results: The overhead is typically between 15% and 30%, as shown in Figure 14(a). Even when the overhead reaches 30%, the overhead is still substantially lower than in the strawman solution that places all rules at the first hop [7, 17]—for example, we distribute a policy of 117.5k rules using 74 switches with 2.7k rules, while the first-hop approach needs 32 edge switches with 17k rules. Figure 14(b) shows the running time of our algorithm, broken down into solving the LP and applying the path heuristic. The LP solver introduces a small overhead, and the path heuristic is responsible for the longer delays. Fortunately, the path heuristic can easily run in parallel, with different threads or processes computing rule placement for different paths. Each pair of bars in Figure 14(b) compares the running time when using one vs. two processes. The speed-up is significant, except for some policies that have one particularly hard path problem that dominates the running time. The algorithm is fast enough to run in the background to periodically reoptimize rule placements for the entire network, with the incremental algorithm in Section 5 handling changes requiring an immediate response.

Evaluating the path heuristic: We also evaluate the path heuristic in isolation to better understand its behavior. These experiments apply the entire endpoint policy to one path of a given length. Figure 15(a) plots the rule-space overhead (as a function of path length) for three representative policies (with the lowest, median and highest overhead) from the university firewall data. The median overhead for the 8-hop case is approximately 5% and the worst case is around 28%. For all policies, the overhead grows steadily with the length of the path. To understand the effect of path length, we compare the results for four and eight switches in the median case. With eight switches, the number of rules per switch (991) is reduced by 44% (compared to 1776 for four switches). But, this also means we must search for smaller rectangles to pack rules into smaller tables. As each rectangle becomes smaller, a rule in the endpoint policy that no longer "fits" within one rectangle



(a) A large university network data



(b) CDF of all test cases

Figure 15: The performance of the path heuristic.

must be split, leading to more extra rules.

Figure 15(b) plots the CDF of the overhead across both the synthetic and real policies for three different path lengths. While overhead clearly increases with path length, the variation across data sets is significant. For 8-hop paths, 80% of the policies have less than 40% overhead, but the worst overhead (from the ClassBench data) is 155%.[11] In this synthetic policy, rules have wildcards in either the source or destination IP addresses, causing significant rule overlaps that make it fundamentally difficult for any algorithm to find good "covering" rectangles. In general, the overhead increases if we tune the ClassBench parameters to generate rules with more overlap. Of the six data sets with the worst overhead, five are synthetic firewalls from ClassBench; in general, the real policies have lower overhead.

Profiling tool: We created a profiling tool that analyzes the structure of the endpoint policies to identify the policies that are fundamentally hard to distribute over a path. The tool searches for all the rectangular predicates in the endpoint policy that can possibly be packed into a single switch. Then, for each predicate q, the tool analyzes the cost-effectiveness ratio between the number of internal rules with respect to q and $\|E_q\|$. The largest ratio here correlates well with the performance of our algorithm. Our tool can help a network administrator quickly identify whether distributed rule placement would be effective for their networks.

[11]Even for this worst-case example, spreading the rules over multiple hops allow a network to use switches with less than one-third the TCAM space than a solution that places all rules at the ingress switch.

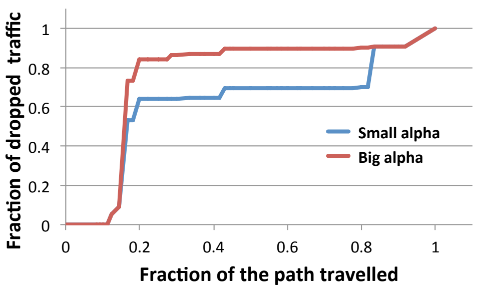


Figure 16: CDF of dropped traffic in the graph

6.3 Minimizing Unwanted Traffic

We next evaluate how well our algorithm handles unwanted traffic (*i.e.,* packets matching a "drop" rule). When ingress switches have sufficient rule space, our LP automatically finds a solution that does not use internal switches. But, when switches have small rule tables, some rules must move to interior switches, causing unwanted packets to consume network bandwidth. Our goal is to drop these packets as early as possible, while still obeying the table-size constraints. We summarize our results using a cumulative distribution function $F(\cdot)$, *e.g.,* $F(0.3) = 0.65$ means that 65% of the unwanted packets are dropped before they travel 30% of the hops along their associated paths.

We evaluate the same test cases in Section 6.2 and assume the unwanted traffic has a uniform random distribution over the header fields. Figure 16 shows a typical result. We run the algorithm using two sets of α values. When α values are small, we want to leave as much unused space as possible; when α values are large, we allow the algorithm to consume more of the available rule space, allowing the path algorithm to drop unwanted packets earlier. In both cases, more than 60% of unwanted packets are dropped in the first 20% of the path. When we give the LP more flexibility, the fraction of dropped packets rises to 80%. Overall, we can see that our algorithm uses rule space efficiently while dropping unwanted packets quickly.

6.4 Comparison with Palette

We next compare our algorithm with Palette [21], the work most closely related to ours. Palette's main idea is to partition the endpoint policy into small tables that are placed on switches, such that each path traverses all tables at least once. Specifically, Palette consists of two phases. In the first phase (coloring algorithm), the input is the network structure and the algorithm decides the number of tables (*i.e.,* colors), namely k, needed. This phase does not need the endpoint policy information. In the second phase (partitioning algorithm), it finds the best way to partition the endpoint policy into k (possibly overlapping) parts. This phase does not require the knowledge of the graph.

Here we compare the performance of our work and

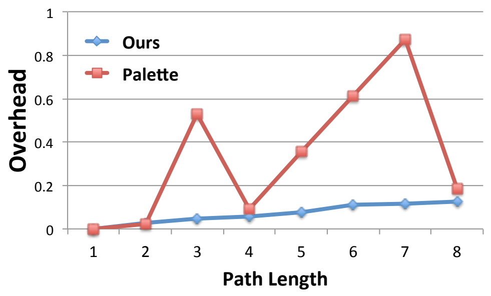


Figure 17: Comparing our path heuristic to Palette

Palette for both the special case where the network is a path and the general graph case. When the network is a path, we only examine the partitioning algorithm (because the number of partitions here exactly equals to the size of the path). Figure 17 shows that Palette's performance is similar to ours when path length is a power of 2 but is considerably worse for other path lengths. Moreover, Palette cannot address the scenario where switches have non-uniform rule capacity.

Next, we examine Palette's performance for general graphs. Specifically, we execute Palette's coloring algorithm on the general graph test cases presented in Section 6.2. The maximum number of partitions found by their algorithm is four. This means in a test case where an endpoint policy contains 117k rules, Palette requires each switch to contain *at least* $117k/4 \approx 29k$ rules (this assumes no further overhead in their partitioning phase). In contrast, our algorithm produces a solution requiring only 2.5k rules per switch.

7. CONCLUSION

Our rule-placement algorithm helps raise the level of abstraction for SDN by shielding programmers from the details of distributing rules across switches. Our algorithm performs well on real and synthetic workloads, and has reasonable running time. In our ongoing work, we are exploring ways to parallelize our algorithm, so we can handle even larger networks and policies efficiently.

8. REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM CCR*, vol. 38, no. 3, 2008.

[3] D. Erickson, "The Beacon OpenFlow controller," in *HotSDN*, Aug 2013.

[4] A. Voellmy and P. Hudak, "Nettle: Functional reactive programming of OpenFlow networks," in *PADL*, Jan 2011.

[5] "POX." `http://www.noxrepo.org/pox/about-pox/`.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, Sep 2011.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *Trans. on Networking*, vol. 17, Aug 2009.

[8] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Hot-ICE*, Mar 2011.

[9] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for data centers," in *CoNEXT*, Dec 2012.

[10] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *PRESTO*, ACM, 2010.

[11] S. Shenker, "The future of networking and the past of protocols," Oct 2011. Talk at Open Networking Summit.

[12] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *NSDI*, Apr 2013.

[13] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *ACM-SIAM SODA*, pp. 1066–1075, 2007.

[14] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 490–500, Apr 2010.

[15] K. Nanxi, L. Zhenming, R. Jennifer, and W. David, "opt." `www.cs.princeton.edu/~nkang/tech.pdf`.

[16] 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.*, vol. 20, pp. 488–500, Apr 2012.

[17] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *CCS*, (New York, NY, USA), pp. 190–199, ACM, 2000.

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

[19] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *ACM SIGCOMM*, (New York, NY, USA), pp. 351–362, ACM, 2010.

[20] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "VCRIB: Virtualized rule management in the cloud," in *NSDI*, Apr 2013.

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

[22] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *ACM SIGCOMM*, pp. 147–160, 1999.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *ACM SIGCOMM*, pp. 213–224, 2003.

[24] S. Suri, T. Sandholm, and P. Warkhede, "Compressing two-dimensional routing tables," *Algorithmica*, vol. 35, no. 4, pp. 287–300, 2003.

[25] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[26] V. Vazirani, *Approximation Algorithms*. Springer, 2004.

[27] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *ACM SIGCOMM*, pp. 207–218, Aug 2010.

[28] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification," in *ACM SIGCOMM*, pp. 335–346, Aug 2012.

[29] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Dover books on mathematics, Dover Publications, 1998.

[30] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, Aug 2012.

[31] K. Calvert, M. B. Doar, A. Nexion, and E. W. Zegura, "Modeling Internet topology," *IEEE Communications Magazine*, vol. 35, pp. 160–163, 1997.

[32] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE INFOCOM*, 2004.

APPENDIX

A. CORRECTNESS OF PATH HEURISTICS

Let $a \cdot b$ to denote the composed action by applying action a before b on a packet. Then,

$$a \cdot \text{Fwd} = a \quad (3)$$

$$\text{Fwd} \cdot a = a \quad (4)$$

LEMMA A.1. *The path heuristics correctly distribute the endpoint policy.*

PROOF. We prove the lemma by induction on the path length. For simplicity, we assume only selecting one rectangular predicate for each switch.

- Base case: path length $\ell = 1$. In this case, we install the entire policy on the only switch, so the heuristics is correct.
- Case: path length $\ell = k + 1$.
  Inductive hypothesis: The heuristics correctly distribute an endpoint policy over a path of length k. Consider selecting a rectangular predicate q on the first switch S_1. Let E be the original endpoint policy and E' be the policy after rewritten. In the heuristics, E_q is the policy(rules) installed on switch S_1. We use $E(\text{pkt}), E'(\text{pkt})$ and $E_q(\text{pkt})$ to denote the action for packet pkt defined by E, E' and E_q separately. Therefore,
  - pkt $\in q$, $E'(\text{pkt}) = \text{Fwd}$ and $E_q(\text{pkt}) = E(\text{pkt})$. $E_q(\text{pkt}) \cdot E'(\text{pkt}) = E(\text{pkt}) \cdot \text{Fwd} = E(\text{pkt})$ by Equation 3;
  - pkt $\notin q$, $E_q(\text{pkt}) = \text{Fwd}$ and $E'(\text{pkt}) = E(\text{pkt})$. $E_q(\text{pkt}) \cdot E'(\text{pkt}) = \text{Fwd} \cdot E(\text{pkt}) = E(\text{pkt})$ by Equation 4;

  By IH, the path heuristics implement E' using the last k switches. Therefore, it implement E using all the $k + 1$ switches.

□

Let $S_1, ..., S_n$ be the ordered switches along a path. Let $R_i (1 \leq i \leq n)$ be the policy(rules) installed on switch S_i.

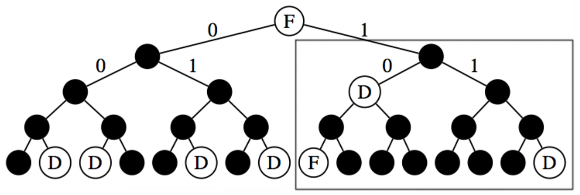
COROLLARY A.2. *For any packet* pkt*, there exists exactly one* $1 \leq k \leq n$*, s.t,* $S_k(\text{pkt}) = Q(\text{pkt})$ *and* $S_j(\text{pkt}) = \text{Fwd}$ *($j \neq k$).*

PROOF. Directly proved from the above lemma. □

LEMMA A.3. *Let* $a_1, ..., a_n$ *be a permutation of* $1, 2, .., n$*. If the switches are arranged in the order* $S_{a_1}, ..., S_{a_n}$*,* $R_{a_1}, ..., R_{a_n}$ *still implement the endpoint policy* Q*.*

PROOF. Consider packet pkt. Let a_k be the switch s.t $R_{a_k}(\text{pkt}) = Q(\text{pkt})$ in Corollary A.2. By Equation 3 and 4,

$$\begin{aligned} & R_{a_1}(\text{pkt}) \cdot ... \cdot R_{a_{k-1}}(\text{pkt}) \cdot R_{a_k}(\text{pkt}) \cdot ... \cdot R_{a_n}(\text{pkt}) \\ = \quad & \text{Fwd} \cdot ... \cdot \text{Fwd} \cdot Q(\text{pkt}) \cdot ... \cdot \text{Fwd} \\ = \quad & Q(\text{pkt}) \end{aligned}$$



(a) Tree representation of an access-control policy

r_1 : (0001 : Drop)	r_4 : (0111 : Drop)	r_7 : (10∗ : Drop)
r_2 : (0010 : Drop)	r_5 : (1000 : Fwd)	r_8 : (∗ : Fwd)
r_3 : (0101 : Drop)	r_6 : (1111 : Drop)	

(b) Prioritized list of rules in a switch

Figure 18: Example one-dimensional policy on a switch

□

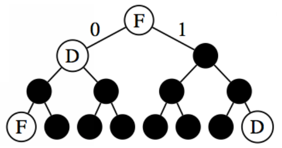
B. ONE-DIMENSIONAL ENDPOINT POLICIES ON A PATH

Along a path, traffic flows in one direction through a sequence of n switches $\{s_1, s_2, ..., s_n\}$, with a link between s_i and s_{i+1} for all $i = 1, 2, ..., n-1$. Since all packets enter the network at the same location, the network-wide policy E depends only on the packet headers; as such, we write $E(\text{pkt})$ instead of $E(\text{loc}, \text{pkt})$. In addition, the routing function R is trivial—all packets arriving at s_1 are routed to s_n, unless E drops the packet. Our goal is to find the minimal capacity C for each of the switch that has a feasible realization of the policy. [12] In this subsection, we consider one-dimensional policies E — those that match on a single header field, such as the IP destination prefix.

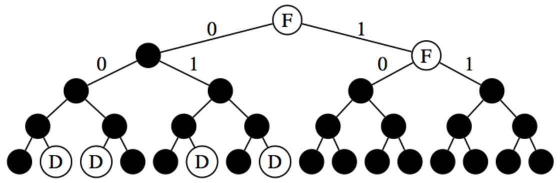
B.0.1 Pack-and-Replace on a Prefix Tree

For one-dimensional policies, there are known algorithms [24] for computing an optimal list of rules for a *single* switch. The function E can be represented as a prefix-tree on the header field, as shown for an example access-control policy in Figure 18(a). Each internal node corresponds to a prefix, and each leaf node corresponds to a fully-specified header value; each prefix can be interpreted as a predicate. A node can specify an action (*e.g.,*forward or drop) for that prefix, as well as descendants that do not fall under a more-specific prefix that includes an action. Computing the prioritized list of rules involves associating each labeled node with a predicate (i.e., the prefix) and an action (i.e., the node label), and assigning priority based on prefix length (with longer prefixes having higher priority), as shown in Figure 18(b). Importantly, one can see that the resulting rules will always satisfy the *nesting property.* In other words, given a pair of rules, their predicates will either be disjoint or one a strict subset of the

[12]There exists a standard reduction between decision problems and optimization problems [25]. So we will switch between these two formulations whenever needed.



(a) Pack operation



(b) Replace operation (without reoptimization)

Figure 19: Pack-and-replace for one-dimensional policies

other—there will never be a partial overlap. Moreover, if rules r_1 and r_2 contain predicates p_1 and p_2, respectively, and p_1 is a strict subset of p_2, then r_1 must have higher priority than r_2. (Otherwise, r_1 would be completely covered by r_2 and would go unused.)

If a single switch cannot store all eight rules in Figure 18, we must divide the rules across multiple switches. Our algorithm recursively *covers* a portion of the tree, *packs* the resulting rules into a switch, and *replaces* the subtree with a single node, as shown in Figure 19. In the "pack" phase, we select a subtree that can "fit" in one switch, as shown in Figure 18(a) (the subtree inside the rectangle). If the root of this subtree has no action, the root inherits the action of its lowest ancestor—in this case, the root of the entire tree (see Figure 19(a)). The resulting rules are then reoptimized, if possible, before assignment to the switch . In this example, switch 1 would have a prioritized list of four rules—(1000, Fwd), (1111, Drop), (10*, Drop) and (1*, Fwd). More generally, we may pack multiple subtrees into a single switch. To ensure packets in this subtree are handled correctly at downstream switches, we replace the subtree with a single predicate at the root of the subtree (e.g., a single "F" node in Figure 19(b)). Then, we can recursively apply the same pack-and-replace operations on the new tree to generate the rules for the second switch.

B.1 Distribute one-dimensional policies

To describe our algorithm precisely, we introduce some notation. Given a switch capacity C, our algorithm decides whether a feasible solution exists. In the "pack" phase, we refer to the subtree packed in a switch as the *projection* E_q of E with respect to a predicate q, defined as follows:

$$E_q(\text{pkt}) = \begin{cases} E(\text{pkt}) & \text{if } q(\text{pkt}) = \text{true} \\ \perp & \text{otherwise,} \end{cases}$$

where $\perp$ refers to "no-operation". As we walk through each switch deciding what rules to place, we maintain an intermediate prioritized list $E' = [r'_1, ..., r'_{n'}]$ to represent the set of unprocessed rules that remain. The set E' starts as E, shrinks as we move along the chain, and ends as a trivial function that forwards all packets.

```
PACK-AND-REPLACE($i, E'$)
1  Let $d_i \leftarrow$ the remaining capacity of $s_i$.
2  $q \leftarrow \arg\max_q\{\|E_q\| \mid \|E_q\| \le d_i\}$
3  Append the rules $E'_q$ sequentially to the end
       of the prioritized list of $s_i$.
4  $E' \leftarrow E' \backslash E'_q$
5  $E' \leftarrow (q, \text{Fwd}) \circ E'$

COMPILE-RULES($\{s_1, ..., s_n\}, E$)
1  $E' \leftarrow E$
2  for $i \leftarrow 1$ to $n$
3      do
4          Add a default forward rule for all
             unmatched packets at $s_i$.
5          while $s_i$ has unused rule space and
             $E'$ is non-trivial
6              do PACK-AND-REPLACE($i, E'$)
```

Figure 20: Path heuristics for one-dimensional policy

Our algorithm is "greedy" in packing as many large subtrees as possible, as early in the chain as possible. At the i-th switch, we recursively perform the "pack-and-replace" operation, and have a default rule that forwards all unmatched packets to the next hop. At each step, we pack the largest possible subtree, subject to the switch's capacity constraint, as shown in the pseudocode in PACK-AND-REPLACE in Figure 20. We pack as many subtrees as possible in a single switch before proceeding to the next switch, as shown in the inner while loop in COMPILE-RULES in Figure 20.

Running time. One can see that a straightforward implementation of our algorithm has time complexity $O(mn \log m)$, where m is the number of rules and n is the number of switches.

B.2 Performance analysis

We have the following theorem regarding the performance of our one-dimensional algorithm.

THEOREM B.1. *Consider our path heuristics described above. We have*

- Correctness of the algorithm*: the prioritized rules in the paths correctly implement the policy function* $E = [r_1, ..., r_m]$.
- Approximation ratio*: let C be the minimum capacity so that our algorithm will return a feasible solution. Let C^* be the optimal capacity. We have $\frac{C}{C^*} \le (1 + \epsilon)$ for any constant ϵ when m and n are sufficiently large.*

PROOF. The first part of the Proposition is proved in

Lemma A.1. We shall focus on the second part. Recall that a lower bound on the optimal capacity C^* for each of the switches is m/n. Thus, we need to show that the capacity C given by our algorithm satisfies $C \leq (1+\epsilon)m/n$, where ϵ tends to zero when $m, n \to +\infty$.

We need the following Lemma.

LEMMA B.2. *Let C be the capacity of each switch. For a switch s_i, the number of prefixes we can find by invoking* PLACE-RULE(s_i, c_i, q_i, E') *is at most* $\log_2 C$.

PROOF. (Proof of Lemma B.2) Let q be an arbitrary prefix. Let q_0 be the prefix obtained by appending a 0 at the end of q (*e.g.,*when $q = 010*$, $q_0 = 0100*$) and q_1 be the prefix obtained by appending a 1 at the end of q. One can see that for any 1-dim policy E,

$$\|E_q\| \geq \|E_{q_0}\| + \|E_{q_1}\| - 1. \tag{5}$$

Let us write c_i be the size of available space for s_i while we are packing subtrees for s_i. We claim that when a q in PLACE-AND-PACK is found and packed, either the size of c_i is reduced by at least a half or $q = *$ (*i.e.,*the prefix represents the whole tree). We use an existential argument to prove the claim. Let q be an arbitrary prefix such that $\|E'_q\| > c_i$ while $\|E'_{q_0}\| \leq c_i$ and $\|E'_{q_1}\| \leq c_i$. Such q always exists unless $\|E'\| \leq c_i$, in which case we complete our argument. Now because of $\|E'_q\| > c_i$ and (5), one of $\|E'_{q_0}\|$ or $\|E'_{q_1}\|$ must be at least $c_i/2$. Thus, the subtree we found that is packed in s_i has at least $c_i/2$ rules. Therefore, the number of prefixes we can find before $c_i = 0$ is at most $\log_2 c$. □

We now continue our analysis by using Lemma B.2. Let us define Φ be the sum of the total number of rules deployed in the switch so far and the size of E', which is a changing variable over the time and is m at the beginning of our algorithm. When our algorithm terminates, one can see that $(n-1)C < \Phi \leq n \cdot C$ (using the fact that all the switches are full except for the last one), *i.e.,*

$$\lceil \frac{\Phi}{n} \rceil = C \tag{6}$$

At the point we deploy rules at the i-th switch s_i, Φ changes in the following way:

- Φ is incremented by 1 because we install a default forwarding rule for all unmatched packets.
- Φ is incremented by 2 when we pack a new subtree in a switch.

Thus, the total increment of Φ at s_i is at most $2\log_2 C + 1$. Therefore, at the end, we have

$$\Phi \leq m + 2(\log_2 C) + n.$$

Together with (6) and the fact that $C^* \geq m/n$, we see that $C \leq (1+\epsilon)C^*$ for any constant ϵ (when m and n are sufficiently large).

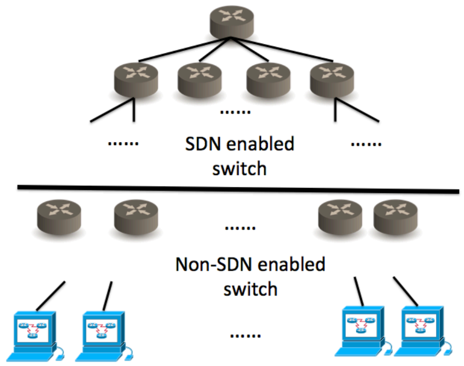


Figure 21: The structure of the load balancer network.

C. CASE STUDY: LOAD BALANCER

This section presents a case study for load balancers. We shall first describe the set up of our network and the design goal of the load balancer. Then we shall describe how our algorithm works here. Finally, we will contrast our algorithm with strawman solutions.

The network and load balancer

In this scenario, we consider a data center that consists of multiple replica servers $\{R_1, R_2, ..., R_n\}$ providing the same servce. Each R_j has a unique IP address and an integer weight γ_i that determines the share of requests the replica should handle. A network of switches is used to connect the servers with the client (See Figure 21). The clients access the service through a single gateway switch. Here, we shall also assume that the distribtuion on the src_ip of the incoming packets is uniform.

The network is a tree and root of the network is the gateway. The tree consists of of two types of switches. Those switches that are close to the root are SDN-enabled switches. Those switches that are closer to the servers are less expensieve, non-SDN-enabled switches. Here we consider an example of the tree that consists of six levels of switches. The first five levels are all SDN-enabled switches and the switches at the last level are non-SDN-enabled switches that directly connect with the servers. Each switch in the first 4 levels contains 4 children. The (SDN-enabled) switches at the 5th level contains 16 children. Each (non-SDN enabled) switch at the 5th level directly connects with 16 servers. Thus, the number of switches at each level is 1, 4, 8, 16, 64, 256, and 4096. The total number of servers is 65536.

Our goal here is to deploy an endpoint policy in the SDN-enabled switches to implement a load balancer so that each server S_i at the end will approximately process γ_i portion of packets. Here, we use a fairly standard recursive algorithm to decide the routing policy (see below for details). Finally, one specific constraint we face here is that the non-SDN enabled switches are less powerful and less flexible to change, thus we require these switches only use destination based routing.

Routing function in load balancer. We now describe our algorithm for building the routing function.The routing function is responsible for making sure each of the server R_j approximately receive γ_j portion of traffic. We use a recursive algorithm to decide the routing function. We explain this idea by demonstrating how it works at the root level. The root (*i.e.,*the gateway) needs to divide the traffic into four ways, each of which goes to one child of the root. Let M be the total number of servers (in our example $M = 65536$). One can see that the portion of traffic that should go to the first child is $\tau_i \triangleq \sum_{1\leq i\leq \frac{M}{4}} \gamma_i$. Similarly, the portion of traffic that should go to the 2nd, 3rd, and 4th children are $\tau_2 \triangleq \sum_{\frac{M}{4}\leq i\leq \frac{2M}{4}} \gamma_i$, $\tau_3 \triangleq \sum_{\frac{2M}{4}\leq i\leq \frac{3M}{4}} \gamma_i$, and $\tau_4 \triangleq \sum_{\frac{3M}{4}\leq i\leq M} \gamma_4$. So the first step of our algorithm is to deploy rules at the root level so that each child receives its corresponding portion of traffic.

Next, we can recursively apply the same algorithm to the children of the root, *i.e.,*for each child, we run this algorithm and make sure their children receive the right portion of traffic. We can continue this recursive algorithm until the routing policies for each of the internal nodes.

Using our solution. The subgraph induced by the SDN-enabled switches corresponds with the big switch abstraction in our solution. Since the non-SDN enabled switches can only use destination based routing, our big switch needs to implement the following two functionalities: 1. *routing:* direct the flow to the right switch; 2. *modification of destination fields:* the dst_ip of each packet needs to be modified to the IP address of the destination server.

Now, these two functionalities together with the load balancing algorithm allow us to define E and R. We may then execute our algorithm to find the rule placements for each switch. Furthermore, notice that under this specific setting, the chain subproblems are all *one-dimensional.* Thus, the overhead variables α_i in the LP are often set to be very small.

Comparison with strawman solutions. We next compare our approach with strawman solutions, highlighting the fact that our algorithm is able to address scenarios that cannot be solved with existing solutions.

There are two trivial (strawman) algorithms, representing two extreme approaches, to modify dst_ip:

- Modify everything at the root (ROOT-SOL): in this case, we need a very powerful switch at the root.
- Modify everything at the last hop in the SDN-enabled switch, *i.e.,*modify everything at the leaves of the big switch (LEAF-SOL). In this case, we need a large number of powerful (but not as powerful as the one used for ROOT-SOL) switches.

In practice, we are often in a world that is between these two extremal spectra, *e.g.,*we have quite a few powerful switches combined with a large number of less powerful switches. The powerful switches are not powerful enough to be placed at the root and modify all the packets. And we do not have a sufficient number of powerful switches so that every leaf on the big switch can get one.

One can see that no trivial solution can address the setting described here while this can be solved by our algorithm in a straightforward manner.

solution param	rules at a leaf	rules at a non-leaf	total rules
total rules	$\perp$	$\perp$	283380
LEAF-SOL	902	19	217844
ROOT-SOL	256	218500	284877
$\beta = 1$	851	851	290191
$\beta = 2$	681	1361	290021
$\beta = 4$	487	1945	289997
$\beta = 8$	310	2477	289905

Figure 22: The structure of the load balancer network.

Results. Here, we assume that are two types of switches to model we have both powerful and less-powerful switches. The switches at the leaf and the non-leaf switches. Notice that there are substantially fewer number of powerful switches (a total number of 85) than non-powerful ones (number = 256). We shall assume that the capacity of the non-leaf switches is β times the capacity of leaf switches. Next, we also assume that the servers are heterogeneous. Each server's processing power is a uniform random variable from $[0, 1]$ (then we need to renormalize the processing powers to get the γ_i parameters).

We test the outcome of our algorithm for the case $\beta = 1, 2, 4$, and 8. We also compare our solution with strawman solutions. See Figure 22. The first row in the table is the total number of rules to represent both the routing and global policy (*i.e.,*the cross product of them). The second and third rows are strawman solutions. The rest of the rows are our solutions. We can see that we are able to find solutions that simultaneously install fewer rules at the leaves than those installed from LEAF-SOL and fewer rules at the non-leaves than those installed from ROOT-SOL. Furthermore, the total number of rules installed from our solution do not change much. Also, notice that the total number of rules given by LEAF-SOL is substantially smaller than the rest of the solutions. This is because we may merge modify and forward actions at the leaves to reduce the amount of rules we need.

C.1 Another interpretation

Let m be the total number of rules that are needed in the cross product between E and R. This number can be computed without knowing the actual capacity of each switch. We notice that regardless how we variate the parameter β, the actual overhead (in terms of total number of installed rules, see the last column in Figure 22) is uniformly small. In fact, after further evaluation, we notice that the total number of rules deployed to the switches do not change much.

In fact, this observation allows us to apply our algorithm in a second way: suppose that a network administrator wants to upgrade non-SDN network into an SDN and the network needs to implement a load balancer. Our solution here can guide the administrator to decide what kind of SDN-enabled switches are needed. Specifically, so long as total number of rules in all the SDN-enabled switches are slightly larger than m, we are able to implement the load balancer.