Raptor: Scalable Rule Placement over Multiple Path in Software Defined Networks

Pravein Govindan Kannan, Mun Choon Chan, Richard T.B. Ma and Ee-Chien Chang
School of Computing, National University of Singapore

***Abstract*—Software Defined Networking (SDN) enables flexible management of networks through policies that are stored as a set of rules in ternary content-addressable memory (TCAM) in the switches. With increasing Cloud and Network Function Virtualization (NFV) demands, the number of network policies required will also increase accordingly. As the TCAM memory capacity available in the switches is limited, any rule placement algorithm must be highly scalable to accommodate a large number of policies in the network. In this paper, we propose *Raptor*, a scalable rule placement scheme that supports *multi-path routing as well as immediate failure-recovery to a backup path without policy violation*. The requirement is that any path which a flow may take should see the same set of policy rules and the challenge is to do this efficiently. We model our problem as an Integer Linear Program to maximize the sharing of rules, followed by heuristic based graph partition for the final placement of rules so as to preserve the rules' priority orderings. Our evaluation shows that *Raptor* can achieve savings of up to 98% in TCAM usage or support 250% more policies compared to existing works.**

I. Introduction

Software Defined Networking (SDN) has revolutionized network programming/configuration by providing an abstract view of the network, thus providing an easier way to express policies. Policies are a set of high-level formal statements that define how packets are processed in a network. The policies are typically translated into prioritized $\{Match, Action\}$ rules that are stored in the ternary content-addressable memory (TCAM) available on switches in the network.

A typical TCAM in a switch can store a few thousand entries/rules [1]. However, with the continuing growth of cloud and network virtualization functions [2], the demand for more and more policies (up to a few hundred thousands to few millions [3]) will increase. Furthermore, as TCAM processing is power-hungry, even with the existing TCAM sizes, it could take up to 60% of the total power consumed by a network switch [4]. Hence, efficient use of TCAM is not only needed because it is a limited resource, but is also needed for reducing power consumption and heat generation [5].

Existing works [3] [6] [7] have looked at distributing policies by distributing over a single path. However, when a link over the allocated path fails or becomes congested (quite frequent in data-centers [8]), the associated rules need to be installed on the fallback path before switching over. This dynamic re-provisioning and placement of rules increase the recovery time as the latency to re-calculate, install and activate these rules can be significant [9] [10].

ISBN 978-3-901882-94-4 © 2017 IFIP

In order to reduce or even eliminate such failover latency, multi-paths are exploited [11]–[15]. The use of multi-path is very common and crucial in traffic engineering to perform load balancing and congestion minimization [16] for meeting the network demands. Existing approaches either (1) duplicate the rules over all possible paths which is inherently expensive [14], or (2) use multi-path only for traffic management and do not duplicate the rules along the path for fast-reroute [11] [12]. As load balancing in high-speed networks demand operation at line-rate, dynamic migration of rules is not practical.

In this paper, we propose *Raptor*, a rule allocation scheme that supports fast-failover/multi-path routing without rule migrations. The key characteristic of our approach is that the set of associated rules are installed on all the paths allocated for traffic between two end hosts. Hence, if one of the paths fails (or is congested), switching over to the other remaining paths can be done quickly. However, catering to multiple paths could lead to excessive replication of rules across multiple paths. Hence, careful placement of rules is needed so that the amount of replication can be reduced.

In order to minimize replication of rules, we leverage rule-sharing, where a single rule's position of placement is determined such that a rule can be shared by multiple paths across the network thus reducing replication. A key observation exploited in our work is that there is a substantial amount of shared rules/policies. Prior studies [1] [17] have shown the existence of substantial wild-card rules which may be shared among various parts of the network.

Raptor consists of the following key modules :

1) **Diffuse:** We formulate the problem as an Integer programming problem that tries to maximize rule sharing. The solution, however, does not necessary preserve rule ordering.

2) **Connect:** Based on the placement generated by Diffuse, we use a graph partition algorithm to distribute the rules to ensure that the ordering between rules is respected.

Evaluation shows that even when only a single path is used, our approach requires 25% to 97% (83.4% average) less rules in the network compared to state-of-the-art (OBS [7]). The performance gap is even bigger when the number of paths increase. If two alternate paths are required, the reduction is 55% to 98% (86.8% average) respectively.

The reduction in TCAM usage can come at a price of increase in network traffic compared to rule-placement on ingress switches. When the objective is to only reduce TCAM usage, the average increase in traffic overhead is 22%. In the case whereby the increase in traffic overhead is restricted to

less than 0.1%, Raptor is still able to reduce TCAM usage by 42% to 93.7% (75.3% average).

The paper is organized as follows: Section II discusses the works that are closely related to *Raptor*. Section III explains the basics of the algorithm. Section IV discusses the algorithms involved in our placement scheme. We present evaluations of our system in Section V. Section VI presents discussions of our work and we conclude in Section VII.

II. Related Work

The most common placement of firewall policies are on the ingress/edge switches [18]. However, restricting the rule to only ingress switches cannot scale to support a larger number of policies. DIFANE [19] implements the policies by partitioning them to the authority switches. The ingress switches redirect the first packet to dedicated authority switches, while caching the rule for future use. vCRIB [3] works on offloading the rules by similar partitioning method from virtual switches to physical switches along the path optimizing the CPU memory of the virtual switches and minimizing traffic overhead due to offloading of rules to physical switches. Palette [6] optimizes the placement of rules given a set of end-point policies by partitioning of the rule graph (constructed by dependency relations) and placing them over the flow-path using Rainbow Path coloring. One Big Switch [7] performs placement of rules by abstracting the network as a single switch and performing placement of rules based on the single path specified for the flow. For each path, rules are grouped in a way to minimize overhead and placed on the switches along the path. Huang et al. [20] optimize the rule placement based on the QoS constraints. They consider rule multiplexing where a set of rules apply to a single flow taking multiple paths. However, they do not perform rule sharing across multiple flows. *Raptor* differs from the above rule placement algorithms in the following ways :

- It Performs rule sharing in cases where a rule might be applicable to the traffic between multiple endpoints in the network.
- It optimizes rule placement considering multiple paths for a particular flows according to the given routing-policy to support fast-reroute and load balancing.
- It partitions the rule-set without causing major overhead. Raptor relies on rule-ordering and packet-tagging to preserve policy semantics (Section IV).

III. Overview

We model the network as a collection of three components as in Figure 1, namely: end hosts $\{H_1,H_2...H_m\}$, switches and links. The end hosts are typically servers or VMs running on top of servers. The switches are further classified into edge switches and core switches. The edge switches $\{E_1,E_2..E_k\}$ ($k \leq m$) connect the (end) hosts to other entities in the network. The core switches $\{S_1,S_2..S_j\}$ are connected to either the edge switches or to the other core switches.

Endpoint Policy: Similar to the assumptions made in OBS [7], we differentiate between policies that determine packet processing (endpoint policies) and policies that determine routing (routing policies). An endpoint policy is a prioritized set of rules which dictate the action taken for each packet entering the network. The policy can be roughly categorized into different categories that stipulate access-control, statistics, QoS provisioning (queues) and other packet re-write actions (e.g. NAT,VLAN, etc). Endpoint policies are applied to all ingress packets before moving through the network switches to the destinations. On the other hand, routing policies dictate the route/path taken by a packet belonging to a flow from one edge switch to the other. The objective is driven by traffic engineering goals, including reduction of loss and latency. Between each host pairs, there are a set of well-defined (chosen) unique paths which the packets can be routed to.

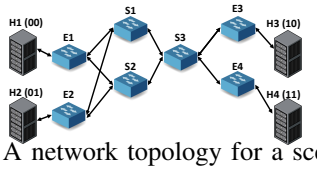


Fig. 1: A network topology for a scenario.

Network policies are typically translated into the appropriate $\{Priority, Match, Action\}$ rules that are stored on the TCAM available on switches in the network in the top-down order of priority. As a host is connected to the network through an edge switch, a rule associated with a host-pair is also associated with a pair of edge switches (E_n, E_m). A rule that applies to a single edge switch-pair is called a **non-shared rule**. A rule that is associated with multiple edge pairs is a **shared rule**. Typically, this rule may contain wild-cards which makes it applicable to multiple flows originating from different end hosts. An example of a shared rule is {Match:src_ipv4=*, tcp_port:22; Action=deny}. This rule is applicable to packets originating from any host in the network. For shared rules, the (default) ingress placement approach would result in replicating these rules on all the applicable edge switches.

Path Coverage: We define **Path Coverage** δ as an indicator function if a switch s lies in a path i:

$$\delta_{\mathrm{i}}(s) = \begin{cases} 0 & \text{if s} \notin \text{i} \\ 1 & \text{if s} \in \text{i} \end{cases}$$

For example, in Figure 1 there are two paths between E_1 and E_3 namely, i_1 = $\{E_1,S_1,S_3,E_3\}$ and, i_2 = $\{E_1,S_2,S_3,E_3\}$. In that case, $\delta_{i_1}(E_3)$ = 1, and $\delta_{i_2}(E_3)$ = 1. However, $\delta_{i_1}(S_1)$ = 1 and $\delta_{i_2}(S_1)$ = 0. We intend to place the endpoint rules in such a way that, regardless of which path (i_1 or i_2) is taken, a packet should adhere to the same set of endpoint policies.

A. A Motivation Scenario

We discuss a scenario in a campus/data center network using the network topology as shown in Figure 1. Consider four hosts $\{H_1(00),H_2(01),H_3(10),H_4(11)\}$ connected to the network via edge switches $\{E_1,E_2,E_3,E_4\}$. Let us consider a set of rules (endpoint policies) to be applied to the network as shown in Table I. To support load balancing and fast-failure recovery, we consider routing packets via two paths between each edge-switch pair. Let us assume the forwarding/failure rules are setup already.

TABLE I: Sample end point policies

ID	Src	Dest	Action
1	H1(00) and tcp-port:80	H3(10) and tcp-port:80	Drop
2	H2(01)	H3,H4 (1*) and tcp-port:443	Drop
3	All(**)	H4(11)	Add VLAN 2
4	All(**)	All(**) and tcp-port:22	Drop

- **Rule 1** is a non-shared rule applicable only to the traffic between H1 and H3 and, edge (switch) pair is e = (E_1, E_3). The candidate path set (I_e) contains $\{i_1,i_2\}$.To ensure rule-enforcement on both paths with minimal replication, it is ideal to place the rule in switches which are present in both paths, i.e. E_1, S_3 or E_3 otherwise, we would need replication of the rule to enforce on other path.
- **Rule 2** is a shared rule applicable to all packets from H2 to H3 and H2 to H4. Hence, effectively, it involves edge E_2 to $\{E_3,E_4\}$. Ideal Placement of this rule is E_2(ingress edge) or S_3, which is shared by all paths from H2 to H3/H4.
- **Rule 3** is applicable to all packets directed towards Host H4. Hence, it is ideal to place this rule in S_3 or E_4, which is shared by all possible paths from all the hosts to H4.
- **Rule 4** is applicable to all possible combination of host pairs. This rule ideally has to be placed in S_3 and E_1 or E_2. In this case, we cannot avoid replicating this rule, since there no single switch covering all paths.

The total number of rules needed in the network is 5 if we consider rule-sharing between different edge pairs. Without rule sharing, and by just allocating the rules in the ingress of flow, 9 rules are needed. OBS/Palette will incur more than or equal to 9 rules, since they remove rule dependencies by slicing the flow-space between rules thus inducing overhead. This simple example illustrates the savings possible through sharing of rules among different paths. The challenge lies in maximizing sharing while preserving rule priorities.

IV. Algorithm / Model

Given the network topology, the selected paths between edge pairs I_e (equivalent to routing policies), the endpoint policies (prioritized rule sets) and the rule (TCAM) capacity of the switches, the objective is to find an optimal placement of rules in the switches with the following constraints: 1) All the routable paths between edge-switch pair should enforce the same endpoint policy and, 2) The number of rules in a switch should not exceed the switch's TCAM capacity.

We model the rule placement problem as an Integer Linear Program (ILP), considering the placement of every rule. However, incorporating ordering in the ILP significantly increases the computation time needed. Hence, we propose *Raptor*, which comprises of two parts :

1) **Diffuse:** We identify the optimal placement of only the shared rules in the network without rule-ordering guarantees. This phase gives us the placement of the shared rules, where the sharing is maximized.

2) **Connect:** This phase takes the placement of shared rules from the diffuse phase, and tries to incorporate the priority constraints using dependency heuristics.

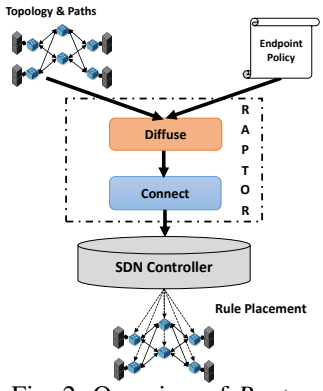


Fig. 2: Overview of *Raptor*.

TABLE II: Notations used in the optimization model

Notation	Description
S	Set of OpenFlow switches.
E	Set of edge switches.
P	Set of shared rules.
e	A pair of Edge Switches.
R_e	Set of shared/non-shared rules for an edge pair e.
C_s	TCAM Capacity of Switch s.
E_ρ	List of edge Pairs to which a rule ρ applies to.
I_e	Set of Candidate Paths between an Edge Pair e.
S_e	Set of Candidate Switches between an Edge Pair e.
L_e	List of edges of the rulegraph for edge pair e.
$O^i_{u,v}$	Ordering between switch u and switch v.
$\delta_i(s)$	The path coverage of Switch s for Path i in I_e.
$D_{\rho,e}$	Switch allocation for rule ρ for Edge Pair e by diffuse.
χ	Direction of Placement, either Forward or Reverse.
$T_{\rho,s}$	Traffic Overhead induced by rule ρ on switch s.

The general overview of *Raptor* is shown in Figure 2.

A. Diffuse

We formulate the problem with the objective to maximize the sharing of rules. Non-shared rules are not considered in the model because : 1) It is restricted to only one edge pair and hence, there is no scope for sharing across multiple end points. 2) It can be flexibly placed along path(s) as long as priority ordering is preserved. We express the objective function as a Boolean allocation matrix denoted by X = ($x_{\rho,s}$) to minimize the total number of rules incurred where $x_{\rho,s}$ = 1 indicates that rule ρ is placed in Switch s and $x_{\rho,s}$ = 0, otherwise.

$$\min : \sum_{\substack{\rho \in P \\ s \in S}} x_{\rho,s} \tag{1}$$

Subject to,

$$\forall s \in S : \sum_{\rho \in P} x_{\rho,s} \leq C_s \tag{2}$$

$$\forall \rho \in P, \forall e \in E_\rho, \forall i \in I_e : \sum_{s \in S_e} \delta_i(s).x_{\rho,s} > 0 \tag{3}$$

Equation 2 accounts for the TCAM rule capacity of the switch. It is generally, good to limit the capacity of each switch to a fixed number of endpoint rules it can hold, so that it can accommodate forwarding rules too. The latest switch architecture [21], [22] support multiple flow-tables/pipeline which facilitates the endpoint and forwarding/routing rules to be placed in separate tables. Equation 3 makes sure that a rule is either placed on a switch which covers all paths in I_e or

multiple switches covering each path in I_e. The value of I_e is provided as input to the ILP Model. Generally, I_e can be decided by the underlying routing module based on the traffic-matrix and bandwidth constraints to load-balance traffic.

*1) **Priority Consideration**:* We derive a model to incorporate ordering between dependent rules with varying priorities to explain the complexity. For each edge-pair e, we construct a dependency rule-graph G_e with the rules R_e (both shared and non-shared rules) applicable for e. Two rules p and q are dependent if a packet header matches both the rules. If two rules p and q are dependent and $p.priority \geq q.priority$, we place a directed edge $l_{p,q}$ from p to q in the graph G_e. Let L_e be the set of all directed edges for e.

We define Ordering $O^i_{u,v}$ between switch u and v corresponding to a path i as below :

$$O^i_{u,v} = \begin{cases} 1 & \text{if u precedes v (or) u=v in path i} \\ 0 & \text{if v precedes u in path i} \end{cases}$$

We formulate the priority constraint for each edge pair as below :

$$\forall e \in E, \forall i \in I_e, \forall l_{p,q} \in L_e : \sum_{u,v \in S_e} x_{p,u}.x_{q,v}.O^i_{u,v} > 0$$

This constraint makes sure that if two rules p and q are dependent($l_{p,q} \in L_e$) and $p.priority > q.priority$, then p would be placed in the same switch or one that precedes the placement switch of q. The complexity of the rule ordering constraint is thus up to $I_e R_e^2 S_e^2$, where I_e is the total number of paths, P is the rule sets applicable to ingress/egress pair e, and S_e is the candidate switches applicable to e.

For increasing number of rules R_e (at least ten of thousands), R_e^2 can easily exceed a billion even for a medium size network. Even if there exists efficient heuristics to solve the ILP, considering ordering constraints will significantly increase the computation time. Hence, we relax the priority ordering constraints and try to find an approximate solution in the diffuse phase and then "correct" the solution for priority ordering as a next step.

The allocation given by Diffuse are the vantage points to place the shared rules in order to maximize their sharing. Based on this solution, in the Connect phase, the final allocation that respects all the constraints including the rule ordering is generated.

B. Connect

In the connect phase, using the output from the Diffuse phase as the reference point, rules are placed along the given paths in order to satisfy ordering constraints.

The overall algorithm is composed of the following steps.

1. Rule Graph Construction: For each edge-switch pair, we form a rule-graph similar to the procedure described in Section IV-A1.

2. Identify Independent Rule-Sets: We identify independent rule sets (disconnected sub-graphs) in the rule graph constructed in Step 1. These independent rule sets can be moved along a given path, without affecting the other rules because they are not dependent on other rules.

3. Placement Initialization: For each edge-switch pair, a pivot switch is chosen. The pivot switch is chosen to be one with most-allocated shared rules in the Diffuse phase. The set of pivots chosen has to cover all paths in I_e. If there is no shared rule, the ingress switch is chosen as the pivot.

Once a pivot is chosen, the initial placement is to put the entire rule set of the edge pair on the associated pivot switch. Since, the entire rule set is present in one switch (pivot), the priority semantics (or rule ordering) are maintained for that edge pair. However, the TCAM capacity may be violated.

4. Graph Partition: In this step, the specific rules are offloaded from the pivot switches such that the priority orderings are respected, TCAM constraints are not violated and rules duplication can be reduced. For ease of presentation, we label the rules belonging to the edge pair with different colors based on the position of the switch:

1) A shared rule is labeled **white**, if it has been allocated to the pivot switch(es) in the Diffuse phase.
2) A shared rule is labeled **red**, if it has not been allocated to the pivot switch(es) in the Diffuse phase.
3) A non-shared rule is labeled **blue**. Note that these rules are not considered in the Diffuse phase because as non-shared rules.

In the heuristics, the white rules placed on the pivot switches are not moved. Only the red and blue rules are considered for relocation. Red rules are moved first because these rules are shared and their placements should be done so that sharing can be maximized. On the other hand, blue rules have no impact on sharing. Their placements are constrained by their relationship with red/white rules and TCAM availability.

Movement of Red Rules: The placement from the Diffuse phase indicates the "ideal" placement for the red rules in order to maximize rule sharing. However, moving a red rule is feasible only if the priority ordering is satisfied with respect to the white rule and availability of TCAM space.

Let $D_{R,e}$ be the position(s) of a red rule R with respect to an Edge pair e in the Diffuse phase. We iterate through the individual red rules and identify whether it shares an edge with a white rule or another red rule. Based on the dependency, and its correctness of the position, it is relocated to $D_{R,e}$.

For example, if a red rule R is dependent on the white rule W (i.e. R→W exists), then $D_{R,e}$ must precede pivot switch in-order to relocate R to $D_{R,e}$ and vice-versa. However, when a red rule R is inter-dependent on two or more white rules (W→R→W), then R will remain at pivot. In addition, when a red-rule is converted to a white rule, it will satisfy the priority ordering for the current edge-pair. However, this particular movement may not hold priority semantics true for other edge-pairs which the red rule applies to. Hence, in cases whereby such a movement causes a priority violation, we slice the flow-space of the red-rule such that it is applicable only for the current edge pair. Now since the red-rule R is no longer a shared-rule, it will be colored to blue, and will be handled as in the next section.

Movement of Blue Rules: The rationale behind moving the blue rules is to utilize the TCAM space of other switches

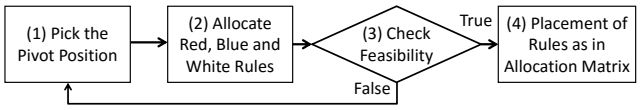


Fig. 3: Overall algorithm flow of connect.

along the path, and also to maintain the rule ordering semantics with respect to re-located red rules. We consider blue rules as a dependency graph (creating a blue cluster), and check their dependencies with a red/white rule(s). Based on their dependency, the placement is restricted to a particular region in the path. For each blue cluster B, let p denote the set containing the positions of red/white rules from which there are incoming edges to B (W→B and R→B) and let n denote the set containing the positions of red/white rules that share an outgoing edge from B (B→W and B→W). Then the blue cluster B is constrained to the switches between positions $\{Max(p),\ Min(n)\}$ for its allocation/movement. $Max(p)$ returns the switch that is farthest from the ingress switch in the list p, and $Min(p)$ return the switch that is nearest to ingress switch in list p.

For each blue cluster, after identifying the region of movement, we place them on switch(es) such that : 1) internal dependencies within the blue rule group are correct and 2) minimize the replication of blue rules when placing them for multi-path. We implement a greedy algorithm($AllocateBlue$) for the allocation of blue rules within a region. Algorithm 1 shows the pseudocode of $AllocateBlue(RuleList, start, end, I_e)$, where $\{start,end\}$ is the range of the switches for allocation of $RuleList$ L derived using its dependency after topological sorting the dependency graph.

The algorithm takes several rounds and during each round, the initial position Q and direction of placement χ is chosen based on the below conditions:

1) If (Cumulative vacancy of switches *after* $Q) > L.size$ Then ; Direction of placement χ is forward.
2) If (Cumulative vacancy of switches *before* $Q) > L.size$ Then ; Direction of placement χ is reverse.
3) If both the above conditions satisfy, Select Direction where cumulative vacancy is highest.

The cumulative vacancy is checked to guarantee feasibility for the rules movement, once a position Q is chosen. The initial position is chosen top down from the switch list (containing candidate switches S_e) which is sorted in decreasing order of the number of paths it covers in I_e. This is done to give preference to switches which cover more paths, and thus reduce the need for replication of the rule(s). Algorithm 1 shows the pseudo code of the $AllocateBlue$ algorithm.

In the event of failure to allocate a rule due to capacity constraints, the current partitioning is discarded and the estimation of pivot (and re-coloring) is repeated again with the rest of the switches. We try to allocate until all pivots are exhausted. If it is impossible to allocate with all pivots we terminate our algorithm. Since, we start with the pivot position which maximizes the sharing of rules, we tend to align towards the optimal solution at best-effort. The overall flow of connect phase for each edge pair is depicted in Figure 3.

Algorithm 1 Allocation for each Blue Cluster

```
function AllocateBlue(RuleList, start, end, I_e)
    Q, χ ← Estimate Initial Position & Direction of Allocation
    if Q is null then
        return false
    end if
    if (χ == Forward) then
        Allocate Decreasing order of priority till vacancy in P.
    else
        Allocate Increasing order of priority till vacancy in P.
    end if
    for Each i ∈ I_e do
        if (δ_i(Q) == 0) then
            AllocateBlue(AllocList, start, end, i)
        end if
    end for
    if (RuleList != Empty) then
        if (χ == Forward) then
            AllocateBlue(RuleList, Q, end, I_e)
        else
            AllocateBlue(RuleList, start, Q, I_e)
        end if
    end if
    return true
end function
```

In order to maintain the one-switch abstraction, we apply a 1-bit tag (T-bit) to the packet which has been matched of a rule, so that it will not be matched to another rule down the path. A rule of the highest priority is added to each switch to match only the T-bit to allow the packet pass down the path stated by the routing policy.

Remark: Note that *priority preservation is always guaranteed* if the Connect phase completes successfully. This is because for each edge-pair, we check the dependency relation of each (red/blue) rule. The movement of red/blue rules is performed with respect to priority dependency with white rules(at pivot) and the priority dependencies of the white rules are preserved since they are on the same switch. If the Connect phase terminates without violating the TCAM constraint, priority ordering is preserved for each edge-pair, and thus for all network policies.

C. An Example Scenario

We illustrate an example of the connect phase in Figure 4 consisting of five rules applicable for edge pair e (S1,S6) along two paths as in I_e with variable TCAM capacity(C) per switch. Rules 1, 4 and 5 are shared rules, and the rest are non-shared rules. Let rules 4, 5 be allocated to S6 by Diffuse, and rule 1 be allocated to S1. Rules 2 and 3 do not have any allocation yet because they are non-shared rules.

1) The pivot is chosen as S6 because it is allocated the highest number of shared rules by Diffuse. Now, the rules are colored accordingly (4 and 5 as white, 1 as red and the rest as blue).
2) We perform the movement of Red rules. Red rule 1 respects its priority relation with white rule 5, by moving it to its ideal position (S1). At the same time, this movement also guarantees feasibility of allocation (TCAM Space) for it and dependent blue rules.
3) The blue rules are moved as a cluster (rule 2 and 3 combined). Since the blue rule cluster has an incoming edge

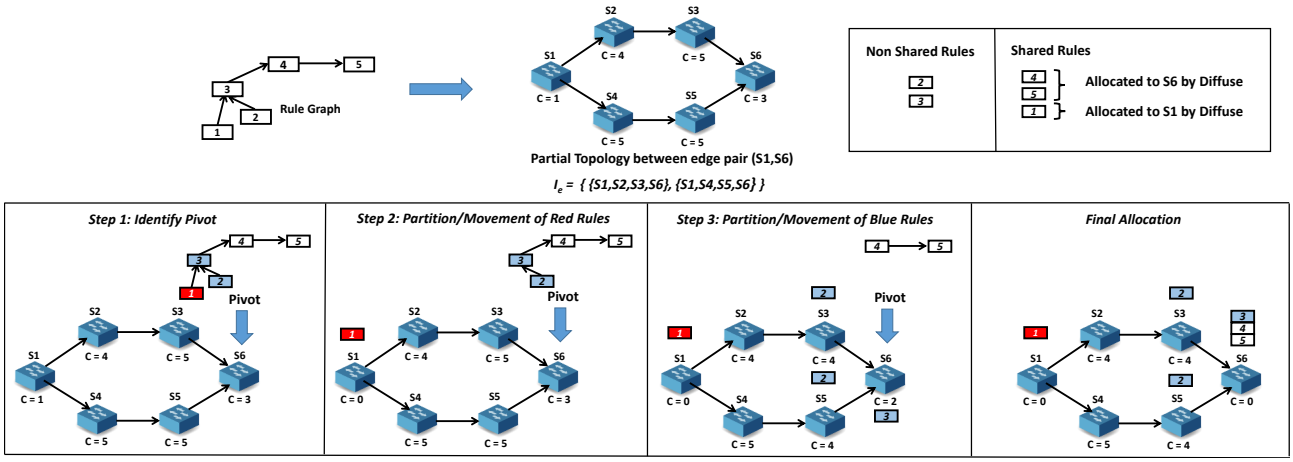


Fig. 4: An example allocation in connect phase

from red rule 1 and outgoing edge to white rule 5, the feasible region of movement is {S1,S6}. We run the *AllocateBlue* algorithm for the blue rule graph, which selects S6 as the Initial Position Q, and places the rules in reverse direction. This would place rule 5 at S6 and rule 4 at S3 and S5. This is the best allocation with least replication in order to provision policies for both paths in I_e. This way, The final rule placement is generated.

D. Dynamic Policy Updates

Network Policies may change from time to time. A policy change may result in an addition of a new rule, removal of an existing rule or update of an existing rule.

1) **Addition of new rule:** First, we must check which end-points (edge switches) the new rule p applies to. There are two cases:

(a) Rule Applies to Single Edge Pair: If the rule p applies to only packets flowing across a single edge pair (non-shared rule), the rule is added to the rule graph of the edge pair. Once added, the rule is considered as a blue-rule, which is allocated as per the procedure mentioned in the previous section.

(b) Rule Applies to Multiple Edge Pairs: If the rule p applies to packets flowing across multiple edge pairs (shared rule), then we apply the diffuse algorithm to this rule, and identify the best initial placement. Once the initial placement is obtained, this rule is added to the existing rule graph for each edge pair. If the dependency conditions hold true, the new policy is allocated to the position stipulated by Diffuse. If any condition fails to be satisfied, we need to break down the rule into individual blue rules and handle it as in Case (1) for each edge pair. If the rule has no feasible placement due to TCAM constraint, we need to run the allocation algorithm (connect) by choosing the pivot position again for the edge pair which failed to given an allocation.

2) **Removal of a rule:** The rule can simply be removed from all the positions of replication using cookie as identifier of the rule. The rule is then removed from the rule graph of the applicable edge pairs similarly to keep it consistent for future updates.

E. Traffic Overhead

One side-effect of spreading the rules along the path in *Raptor* is the increase of traffic overhead, due to movement of ACL/QoS rules further down the path. Thus, packets are forwarded inside the network and are dropped later along the path. The amount of traffic overhead incurred can be incorporated into the objective of the ILP Model (Diffuse). We define the overall overhead incurred due to the placement of an overhead inducing rule ρ (deny, rate-limiting rules) on switch s as :

$$T_{\rho,s} = \sum_{\substack{i \in I_e \\ e \in E_\rho}} \sum_{u,v \in S_e} \tau_\rho^i . h_s^i \tag{4}$$

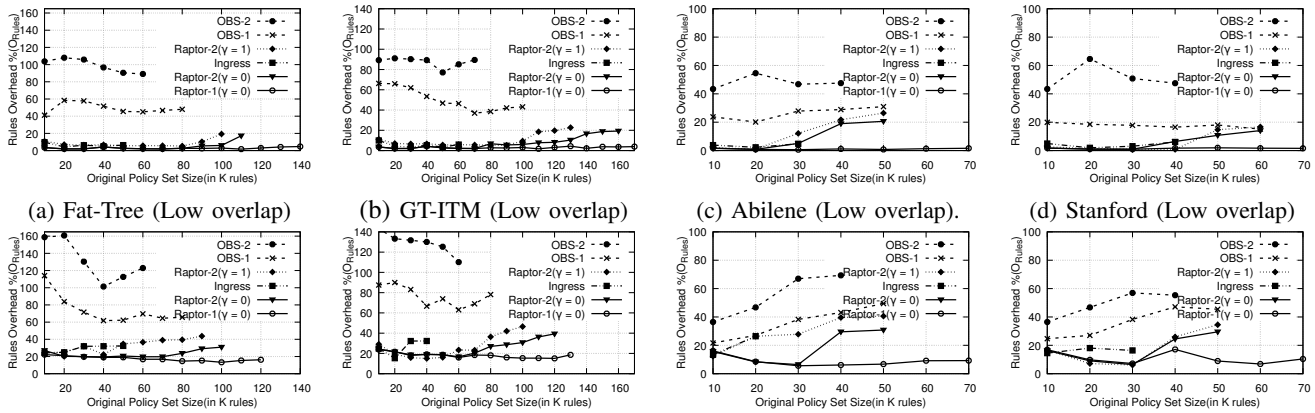
where, τ_ρ^i is the traffic weight matching rule ρ on a path i and, h_s^i is the hop-count of switch s from the ingress switch of a path. For non-overhead inducing rules (accept, modifying rules), τ_ρ^i = 0. We modify the objective function of the ILP Formulation in the Diffuse phase (Equation 1 from section IV-A) as follows to include traffic overhead variables :

$$\min : \sum_{\substack{\rho \in P \\ s \in S}} ((T_{\rho,s}.\gamma) + 1).x_{\rho,s} \tag{5}$$

where γ is a tuning parameter which can be set to any value between 0 and 1. A value close to 1, will make sure the model gives importance to reducing traffic overhead, and a value close to 0, will make sure the model gives importance to reducing overall rules incurred.

V. Evaluation

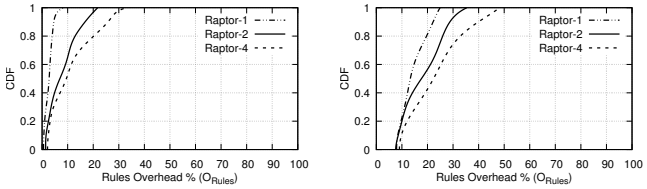
We have implemented *Raptor* in Java, and used Gurobi [23] as the ILP solver. Additionally, we integrated *Raptor* with the ACL module of the Floodlight [24] controller. We used the Floodlight's feature of accessing pairs to determine which edge switch an end-host is connected to. This information is used to determine if a particular IP-prefix in a rule is associated with a single switch (non-shared rule) pair or multiple switches (shared rule). We evaluate *Raptor* over 4 network topologies: 1) **Fat-Tree8** with 80 switches, 2) **Structured Networks** with 100 switches based on GTM-ITM [25], 3) **Stanford Backbone Network** (25 switches) [26] and, 4) **Abilene** [27] (39 switches).



(a) Fat-Tree (Low overlap) (b) GT-ITM (Low overlap) (c) Abilene (Low overlap). (d) Stanford (Low overlap)

(e) Fat-Tree (High overlap) (f) GT-ITM (High overlap) (g) Abilene (High overlap) (h) Stanford (High overlap)

Fig. 5: Average Overhead (O_{Rules}) in various topologies with low overlapping rules(policies) and High overlapping rules.



a. Low overlap b. High overlap

Fig. 6: *Raptor* with path coverage: 1, 2 and 4

The policies are generated by ClassBench [28] with twelve different ACL seeds. We classify the policy sets into two variants: 1) Low overlap (<7% shared rules) and, 2) High overlap (∼20% shared rules). The evaluations were run on a Dell Optiplex 9020 with 8-core i7-4790 CPU@3.60GHz and 32GB of RAM. We vary the size of the policy sets starting from 10K till a feasible solution cannot be found for the given algorithm and network topology. We compare *Raptor* against the two rule allocation schemes:

(1) Ingress [18] places rules at the edge switches without creating additional rules due to splitting/slicing but will fail if the TCAM space on the ingress switches are exhausted.

(2) OBS [7] is an existing rule placement algorithm proposed by Kang et al. We do not consider Palette [6] for our evaluation because it performs equal or worse than OBS [7]. Also, we do not compare with DIFANE [19] since their objective is to minimize controller overhead using authoritative switches with high TCAM capacity.

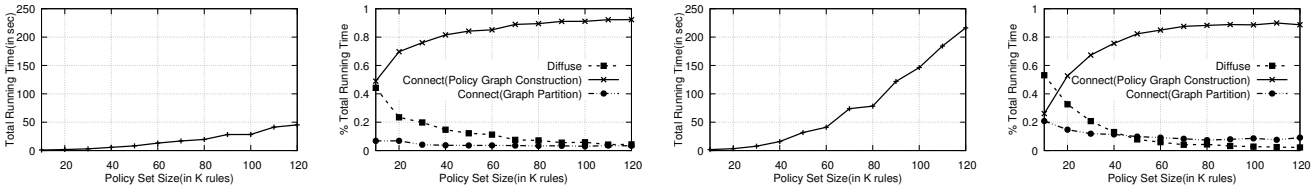
The multiple path versions of our scheme will be labeled as *Raptor*-k, where k is the number of paths available between a given end-host pair. These k paths are link/edge distinct and computed using the Suurballe's algorithm [29]. Traffic overhead minimization version of *Raptor* will be labeled as *Raptor*-k(γ = 1). Recall that, with a high $\gamma = 1$, the diffuse model tries to minimize the traffic overhead over number of rules generated. About 50% of the policy-set is set to contain overhead-inducing rules. For comparison, we extend the OBS algorithm to support $k > 1$ paths and is called OBS-k. In the OBS-k allocation, each rule should appear in the k paths at least once. Each switch maintains a TCAM capacity of 3500. The metric used in the evaluation is the *percentage increase in the number of switch rules* incurred to install the policies in the network computed as $O_{Rules} = (Res/Pol - 1) * 100$, where Res is the number of rules generated by the given algorithm and, Pol is the number of rules specified in the given policy set. Each data point shown in the evaluation is the average of 6 runs (6 different ACL seeds). We refer to *Raptor*-k(γ = 0) as just *Raptor*-k throughout this section for simplicity.

A. Overhead Comparison

We evaluate the efficiency of the various algorithms by looking at the overhead O_{Rules} for the 4 different topologies and 2 sets (low and high overlap) of rules. The results for the low and high overlap rule sets are shown in Figure 5(a) to 5(d) and Figure 5(e) to 5(h) respectively.

When the percentage of shared rule is small, the Ingress algorithm is very efficient and can be accommodated on the edge switches. Note that due to the shared rules, the overhead is still not zero since the shared rules need to be replicated. The Ingress allocation fails to scale beyond 60K rules for all topologies. For all cases, *Raptor*-1 has the lowest overhead and can support the largest number of rules. The overall savings of *Raptor* compared to OBS-1 is 25% to 97% (83.4% on average) with a higher saving when rule-overlaps are higher. In terms of topology, the improvement of *Raptor* over the other algorithms is lower in the Stanford and Abilene network because the networks are smaller and the topologies more restrictive (OBS' overhead is low with less no. of hops).

When the number of paths is increased from 1 to 2, the incremental increase in overhead is relatively small for *Raptor*. Over all cases, the increase ranges from 0% to 19% for low overlap and 0.1% to 24% for high overlap. Even though *Raptor*-2 needs to allocate policies/ rules along two paths, it is able to outperform OBS-1 and OBS-2. This is because *Raptor* is able to reduce the overhead by placing the shared rules on switches in overlapping paths. On the other hand, when OBS is required to support two paths (OBS-2), the overhead (O_{Rules}) increases by nearly 100% in many of the cases. This is because the sharing of rules across paths is not taken into effect in OBS. OBS does not specifically have a trend in the rules overhead upon increase in policy-set because its LP Solver



a. Running Time(Low overlap) b. Ratio(Low overlap) c. Running time(High overlap) d. Ratio(High overlap)

Fig. 7: Running Time analysis of *Raptor*.

accepts any solution that is feasible. In the case of *Raptor*, we see a consistent increase in the overhead with larger policy-sets because the sharing is reduced as the TCAM space in the optimal locations have been exhausted. Overall, *Raptor*-2 achieves 55% to 98% (86.8% on average) reduction over OBS-2. A direct consequence of this saving can be observed by the ability of *Raptor*-1 and *Raptor*-2 to support more rule sets than the other schemes. The results in Figure 5 also show the range of rule set sizes that the different algorithms are able to support. *Raptor*-1 is able to accommodate 30% to 70% more rules over OBS-1, and *Raptor*-2 is able to accommodate 25% to 125% more rules over OBS-2. Finally, *Raptor*-2 supports 100% to 250% more rules compared to the Ingress allocation.

With *Raptor*-2(γ = 1, minimize traffic overhead), we see an increase in rules overhead compared to *Raptor*-2 as the algorithm tries to place the traffic overhead inducing rules towards ingress. Nevertheless, the TCAM usage is still lower than the Ingress allocation. We will discuss the efficiency of traffic overhead minimization further in subsection V-D.

B. Increasing Number of Paths Available

We vary the number of paths to be supported from 1 to 4 between each end-hosts and observe the percentage rule overhead incurred. Due to the need to have sufficient link/edge distinct paths, this evaluation only uses the Fat Tree topology. The results for *Raptor*-1, *Raptor*-2 and, *Raptor*-4 are shown in Figure 6. The figure shows the Cumulative Distribution Function (CDF) of O_{Rules} over 60 different policy sets. We observe that, while the overhead increases with more paths, the increment is relatively small. This is because *Raptor* is able to exploit the multi-path characteristics of the topology (nodes overlapping on multiple paths) for the ideal placement of rules. For the rule sets with low overlap, the 80^{th} percentile overhead values for *Raptor*-1 and *Raptor*-4 are 4% and 20% respectively. Similarly, for the rule sets with high overlap, the 80^{th} percentile overhead values for *Raptor*-1 and *Raptor*-4 are 19% and 30% respectively. *Raptor* does not induce any changes in the latency/ throughput of the flow, since routing policies are not modified. However, endpoint policies are applied at various points along the path.

C. Running Time

We measure the running time for *Raptor* to compute a feasible allocation. We group our results according to the policy sets of two variants : a) Policy sets with less overlapping/shared rules, and b) Policy sets with high overlapping/shared rules. Figures 7(a) and Figure 7(c) show the average running times for various policy set sizes for the

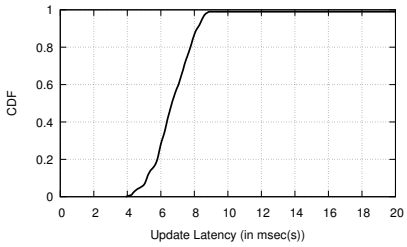


Fig. 8: Running time distribution of updates

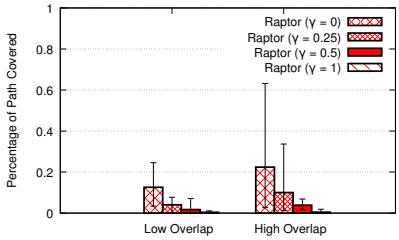


Fig. 9: Traffic Overhead due to γ variants of Raptor

Fat-Tree topology. For policy sets with less shared rules, computation completes in less than a minute even for up to 120K rules. For policy sets with more shared rules, the computation time increases to up to 2.5 minutes for 120K rules. To understand the computation time better, we break down the running time into three major components:

1) Diffuse (ILP Solver)
2) Connect (Rule graph construction & Rule set Identification) : Steps 1-3 in Section IV-B.
3) Connect (Graph partition): Step 4 in Section IV-B.

Figures 7(b) and 7(d) show the ratio of time taken by each component with respect to the total running time. We note that rule graph construction and rule set identification constitute the major part of the running time at around 80-90% in most of the cases. The complexity of graph construction is $O(E.P^2)$, where P is the number of rules for each edge pair in E. In practice, the entire computation only needs to be executed infrequently. For policy updates that involve only a small number of the rules, the computation is much faster. Also, we could parallelize the construction for each edge pair, as they are independent graphs. In particular, addition of a new rule is O(P), and removal of a node is O(1). We show the distribution of update latency over about 250K updates in Figure 8. We calculate the time taken by *Raptor* to determine the location/placement of the newly added rule to the set of existing rules. The number of existing rules is varied from 100K to 150K. We observe that the computation takes less than 9ms for 99% of the policy updates and is less than 20ms

in all cases. Hence, the algorithm is fast enough to process at least $\sim$110 updates/sec.

D. Minimizing Traffic Overhead

We measure the overall traffic incurred by our placement scheme in this subsection. We assume that traffic is uniformly distributed among the flows. We perform our measurement on the complete topologies and policysets. We run four variants of *Raptor* : 1) $\gamma = 0$ (Only consider rule minimization), 2) $\gamma = 0.25$ and 3) $\gamma = 0.5$ and $\gamma = 1$ (Minimize traffic overhead). We measure traffic overhead in terms of percentage of the path covered by a traffic matching a certain rule.

We plot our average traffic overhead incurred per rule (only overhead inducing rule) with a uniform traffic distribution in Figure 9. In cases of high overlap, *Raptor* ($\gamma = 0$) induces traffic overhead of 22.4% per rule, which means on an average every overhead inducing rule induces the restricted traffic to come through 22.4% of the entire path. However, with *Raptor* ($\gamma = 1$) the traffic overhead drops to 0.04%. In cases of low overlap rule set, *Raptor* ($\gamma = 0$) induces traffic overhead of 12%. The trend in decrease of traffic overhead is clearly evident with the increase in γ to 1.

Finally, we note that Raptor allows user to trade-off traffic overhead for TCAM usage. In the case where traffic overhead is negligible (0.04%), TCAM usage reduction of 42% to 93.7% (75.3% on average) can still be achieved.

VI. Discussion

Raptor currently performs best to handle link-failure scenarios, since it looks to co-locate flows on common nodes between various paths. In order to handle switch-failures, the paths provided to *Raptor* needs to be node-disjoint. The impact of such a requirement is that rules will need to be replicated. Hence, at the discretion of network-programmer the paths for selective flows could be node-disjoint or link-disjoint.

Raptor looks into the placement of the rules. In order to achieve reduction in the given input rule set, we could additionally use rule compression to group redundant/similar rules to shared rules. We added a compression module before the diffuse algorithm and observed an additional reduction in the Total Overhead(O_{Rules}) by up-to 3% for low-overlapping rules, and up-to 24% for high-overlapping rules.

VII. Conclusion

We have presented an efficient/scalable rule placement algorithm, *Raptor* to optimize the placement for multi-path/fast reroute traffic in Software Defined Networks. We have achieved this by leveraging the sharing of rules across endpoints of the network. We formulate our problem as an Integer Linear program(ILP), and finally, apply heuristics to maximize the sharing, and adhere to priority ordering. We have build Raptor prototype and also integrated it with Floodlight controller. Our Evaluation of Raptor with various topologies and rule sets generate by ClassBench [28] has demonstrated great savings in the TCAM space of the switches.

Acknowledgement: This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2015-NCR-NCR002-001) and administered by the National Cybersecurity R&D Directorate.

References

[1] C.Yu, C.Lumezanu, H.V.Madhyastha, and G.Jiang, "Characterizing rule compression mechanisms in software-defined networks," in *PAM*, 2016.
[2] "NFV Forecast," https://www.sdxcentral.com/reports/ sdn-nfv-market-size-forecast-report-2015/.
[3] M.Masoud, M.Yu, A.Sharma, and R.Govindan, "Scalable rule management for data centers," in *NSDI*, 2013.
[4] "HP Energy Efficient Networking Business Whitepaper, 2011," http://h17007.www1.hp.com/docs/mark/4AA3-3866ENW.pdf.
[5] B. Agrawal and T. Sherwood, "Modeling tcam power for next generation network devices," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, 2006.
[6] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM*, 2013.
[7] N.Kang, Z.Liu, J.Rexford, and D.Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *CoNEXT*, 2013.
[8] P.Gill, N.Jain, and N.Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *SIGCOMM*, 2011.
[9] K.He, J.Khalid, A.Gember-Jacobson, S.Das, C.Prakash, A.Akella, L.E.Li, and M.Thottan, "Measuring control plane latency in sdn-enabled switches," in *SOSR*, 2015.
[10] M.Kuzniar, P.Perešíni, and D.Kostić, "What you need to know about sdn flow tables," in *PAM*, 2015.
[11] C. Hopps, "Analysis of an equal-cost multi-path algorithm," *RFC 2992*.
[12] NKang, M.Ghobadi, J.Reumann, A.Shraer, and J.Rexford, "Efficient traffic splitting on commodity switches," in *CoNEXT*, 2015.
[13] A.Kabbani, B.Vamanan, J.Hasan, and F.Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *CoNEXT*, 2014.
[14] M.Reitblatt, M.Canini, A.Guha, and N.Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *HotSDN*, 2013.
[15] N. L. M. V.Adrichem, F.Iqbal, and F. A. Kuipers, "Computing backup forwarding rules in software-defined networks," *CoRR*, 2016.
[16] K.He, E.Rozner, K.Agarwal, W.Felter, J.Carter, and A.Akella, "Presto: Edge-based load balancing for fast datacenter networks," *SIGCOMM*, 2015.
[17] T.Benson, A.Akella, and D.A.Maltz, "Mining policies from enterprise network configuration," in *IMC*, 2009.
[18] S.Ioannidis, A. S.M.Bellovin, and J.M.Smith, "Implementing a distributed firewall," in *CCS*, 2000.
[19] M.Yu, J.Rexford, M.J.Freedman, and J.Wang, "Scalable flow-based networking with difane," in *SIGCOMM*, 2010.
[20] H. Huang, S. Guo, P. Li, B. Ye, and I. Stojmenovic, "Joint optimization of rule placement and traffic engineering for qos provisioning in software defined network," *Computers, IEEE Transactions on*, 2015.
[21] R.Ozdag, *Intel ® Ethernet Switch FM6000 Series - Software Defined Networking*, 2012.
[22] P.Bosshart, G.Gibb, H.S.Kim, G.Varghese, N.McKeown, M.Izzard, F.Mujica, and M.Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM*, 2013.
[23] "Gurobi Optimization Framework," http://www.gurobi.com/.
[24] "Floodlight Controller," http://www.projectfloodlight.org/floodlight/.
[25] K. Calvert, M. Doar, and E. Zegura, "Modeling internet topology," *Communications Magazine, IEEE*, 1997.
[26] P.Kazemian, G.Varghese, and N.McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.
[27] "Internet2 Topology," https://www.internet2.edu/media/medialibrary/2015/08/04/NetworkMap_all.pdf.
[28] D. J.S.Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, 2007.
[29] J.W.Suurballe and R.E.Tarjan, "A quick method for finding shortest pairs of disjoint paths," *Networks*, 1984.