Consistent SDN Rule Update with Reduced Number of Scheduling Rounds

Mahdi Dolati*† Ahmad Khonsari*‡ Majid Ghaderi†
* University of Tehran, Tehran, Iran; Email: {mahdidolati,a_khonsari}@ut.ac.ir
‡ Institute for Research in Fundamental Sciences (IPM), Tehran, Iran; Email: ak@ipm.ir
† University of Calgary, Calgary, Canada; Email: {mahdi.dolati,mghaderi}@ucalgary.ca

***Abstract*—Consistent operation of software-defined network (SDN) switches during the transient periods of forwarding rule updates is a critical issue. This paper studies the problem of updating SDN rules, while preserving two essential security and performance consistency properties: (1) Waypoint Enforcement which mandates that all packets traverse a specific checkpoint (*e.g.*, firewall), and (2) Loop-Freedom that prevents forwarding packets along a loop. To guarantee these properties, we schedule rule updates in multiple rounds. To reduce the time that the network stays in the transient period of updating the switches, we have to solve the NP-hard problem of minimizing the number of update rounds. To this end, we design a fast algorithm called RRS which can be applied to very large networks. Our experiments on a large dataset of 28K scenarios show that RRS achieves a 323x improvement in the median of execution time compared to solving the exact Mixed Integer Program (MIP) formulation.**

***Index Terms*—Consistent Update, Waypoint Enforcement, Loop-freedom**

I. Introduction

A. Background and Motivation

Software-Defined Networking (SDN) is a network architecture which decouples the network control logic from the underlying switches that forward traffic. The decoupled network control logic is implemented in a logically centralized SDN controller [1]. The controller installs and updates forwarding rules on network switches instructing them how to forward traffic in the network. To maintain an optimal forwarding configuration, the controller updates the forwarding rules frequently, *e.g.*, in response to events such as flow arrivals, congestion, device outages, or end-host migrations [2].

Updating forwarding rules in SDN switches is a challenging task since the delay of sending new rules to switches as well as the time required to install them in the memory of switches are non-deterministic. Therefore, even if the controller sends out the new rules at the same time, some switches may update their forwarding behavior considerably sooner than others [2]. As a result of this asynchronous behavior, there is a possibility that the network state becomes inconsistent during transient periods of rule updates, which results in malfunctions such as sending packets along transient loops [3], overloading some links [4], or bypassing a firewall [5]. Thus, the order in which switches are updated by the controller is important for avoiding inconsistent network states.

The network properties that should not be disrupted throughout the rule update process are called the network *consistency properties*. For example, Waypoint Enforcement (WPE) is the consistency property that requires each packet to traverse specific waypoints, *i.e.*, a middlebox such as a firewall [5]. Modern enterprise networks rely on a large number of in-network functions [6]–[8] to satisfy a variety of requirements such as security and performance [9]. As such, certain middleboxes should process every packet that enters the network [10]. Specifically, WPE is a desirable consistency property in virtualized and security critical environments. Loop Freedom (LF) is another crucial consistency property [11] that guarantees loops do not occur in the network, and is considered an important performance requirement. A rule update is *consistent* if it does not violate the network consistency properties. In this work, we focus on updating forwarding rules, while preserving WPE and LF.

Since the consistency of a rule update in a switch depends on the forwarding state of other switches, the SDN controller should coordinate rule updates across the switches to preserve the consistency properties. Specifically, it should send the updates to switches in multiple rounds in such a way that the order of updates in a single round does not affect the network consistency properties [12]. To reduce the time it takes to complete the update process, *i.e.*, the time that the network spends in an incorrect or sub-optimal configuration [2], it is desirable to design an update process that minimizes the number of update rounds. It has been shown that minimizing the number of update rounds is NP-hard [9], as such, in this paper we focus on designing fast approximation and heuristic algorithms for the problem of consistent SDN rule update with minimum number of rounds.

B. Related Work

A comprehensive survey on consistent network update algorithms is provided in [13]. Although there exist suitable solutions for the problem in traditional networks [14]–[16], SDN requires different solutions due to different network constraints and capabilities. We briefly review the works on consistent SDN rule update that are more relevant to our work.

Update Mechanisms and Objectives. A number of papers consider the basic consistency properties of congestion freedom [4], [17]–[20] and guaranteed packet delivery [11], [21]. These works, however, do not consider preserving higher-level policies that network operators usually demand. Such policies define a set of constraints on the paths a packet can traverse during the update process. To address this shortcoming, in a seminal work, Reitblatt *et al.* [3] proposed the per-packet

978-3-903176-14-0

consistency (PPC) property to ensure that every packet is handled either by the old policy or the new one, but never a combination of the two. Then, they designed a 2-phase commit algorithm, which works based on packet tagging, to enforce PPC. However, the 2-phase commit algorithm doubles the usage of expensive and power hungry TCAM memory in switches, to the point of making it impractical [22]. As a result, Mahajan *et al.* [12] proposed to schedule the rule updates in multiple rounds. Minimizing the required number of rounds is specifically desirable, because it reduces the time that the network remains in a transient state [2]. However, even for the most elementary consistency property, *i.e.*, loop-freedom (LF), not only the problem of minimizing the number of rounds is NP-hard but also it is NP-hard to approximate the number of rounds with an approximation ratio better than $4/3$ [21]. Currently, these exist no approximation algorithm for this problem in the literature [13]. Instead of minimizing the number of rounds, Amiri *et al.* [23] proposed a greedy scheduler which maximizes the number of updated switches in each round. Nevertheless, this new problem is NP-hard and can increase the number of rounds by a factor of $\Omega(|V|)$ [24].

Policy Preserving Properties. PPC is an unnecessarily strong requirement in practice. As such, several subsequent works considered different properties. For example, McClurg *et al.* [25] studied the class of properties that can be defined as a linear temporal logic formula. However, they did not consider the problem of minimizing the number of update rounds. Vissicchio and Cittadini [26] proposed the FLIP algorithm for policy preservation, which guarantees that each flow traverses a set of pre-defined paths in the network. However, FLIP uses packet tagging to improve its performance and consequently increases the memory consumption of the switches. Ludwig *et al.* [5] suggested the waypoint enforcement (WPE) property as a replacement for PPC. Then, they proved the NP-hardness of minimizing the number of scheduling rounds while satisfying the LF and WPE properties and extended WPE to enforce multiple waypoints in the network [9]. Their approach for minimizing the number of rounds relies on solving a mixed-integer program (MIP) which is generally computationally intractable and thus not scalable in practical applications.

C. Our Work

In response to the shortcomings of existing works on preserving policies in SDNs, *i.e.*, solving MIP formulations directly [5], [9], increasing TCAM memory usage [26], and ignoring the number of rounds as the objective [23], [25], we propose an algorithms, called Reduced-Round Scheduler (RRS), for computing efficient update schedules that satisfy LF and WPE. Our contributions can be summarized as follows:

- We design our heuristic based on a critical observation that the consistent rule update problem is *reversible*. We prove the reversibility of the problem by showing that any solution of the backward problem (*i.e.*, the problem of consistently updating the network from the final configuration that the controller desires to deploy, to the current configuration that the controller has decided to change) is also a solution for the forward problem.
- We evaluate the performance of the proposed algorithm on an extensive dataset that contains 28K update scenarios with a varying number of switches.

The rest of the paper is organized as follows. Section II defines the problem. The RRS algorithm is presented in Section III. Evaluation results are presented in Section IV. Section V concludes the paper.

II. System Model and Problem Definition

A. Network Model

We consider an SDN network with a centralized controller and a set of SDN-enabled switches that are connected to the controller. We focus on the unsplittable flow model that restricts a flow to carry its traffic over a *single path*. Although this model makes the problem more difficult to solve, complications such as packet re-ordering, which, for instance, negatively affects TCP performance are avoided. Consider the network depicted in Fig. 1 and assume that the controller needs to update the old path of flow f (represented by solid lines), *i.e.*, $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6)$ to a new path (represented by dashed lines), *i.e.*, $(1 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 6)$. Because updating switches that are not common between the old and new paths of f is trivial [9], we define an induced graph $G(V, E)$ over the set of common switches, where V denotes the set of common switches between both paths, *i.e.*, $V = \{1, 3, 4, 5, 6\}$. For every switch pairs $u, v \in V$, the link (u, v) belongs to E, if 1) there is a direct link between u and v in the original SDN network, or 2) there is a path in the original SDN network between u and v, such that none of the intermediate switches are common between the old and new paths. For example, paths $1 \rightarrow 7 \rightarrow 5$ and $1 \rightarrow 2 \rightarrow 3$ in Fig. 1 are represented, respectively, as links $(1, 5), (1, 3)$ in Fig. 2, which are included in E. Denote the old and new paths in G by π_{old} and π_{new}, respectively, and let s and d denote the source and destination switches. In Fig. 2, π_{old} is $(1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6)$ and π_{new} is $(1 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 6)$. In the rest of paper, we focus only on the induced graph G.

B. Consistency Model

We consider Relaxed Loop-Freedom (RLF), which is a fundamental performance-related consistency property. RLF prevents loops in the routing path of flows. However, since transient loops which are *unreachable from the source switch* have a negligible impact on the network performance [24], RLF allows such loops to happen. Compared to eliminating all loops regardless of their reachability, RLF can accelerate the update process by a factor of $\Omega(|V|)$ [24]. For example in Fig. 2, suppose that only switch 1 is updated and thus the current forwarding path is $(1 \rightarrow 5 \rightarrow 6)$. Updating switch 4 creates the loop $(3 \rightarrow 4 \rightarrow 3)$, however, RLF allows this loop because it is not reachable from the source switch 1. In addition to RLF, we also consider Waypoint Enforcement (WPE), which is a desirable security consistency property. WPE ensures that every packet of each flow goes through

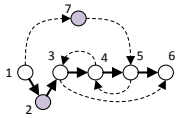


Fig. 1: An update scenario in which the old and new paths are represented by solid and dotted lines, respectively. Switches that are not common between old and new paths are shaded.

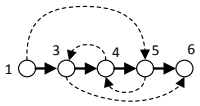


Fig. 2: Induced graph $G(V, E)$ in which the non-common switches are omitted and their attached links are contracted.

a specific node in the network. Throughout the paper, we call this special node the *waypoint* and denote it by wp. Consider Fig. 2 for an example of WPE violation. Let switch 4 be the waypoint. If switch 1 is updated before other switches, traffic will traverse path $(1 \rightarrow 5 \rightarrow 6)$, and hence bypasses the waypoint 4.

C. Update Model

The SDN controller can update the routing path of each flow by changing the rules installed in the memory of the network switches without affecting the routing of other flows. Therefore, we assume that there are no conflicts or dependencies among the rules which we install in a single switch, and consequently we can update a switch in a single *round* of message passing [12]. Thus, we focus on the inter-switch rule dependencies and assume all intra-switch rules are non-conflicting. In each round, the controller updates a subset of switches $U \subseteq V$, where updating switches in U in any order preserves LF and WPE. The controller starts a new round after confirming that the switches chosen in the previous round have installed the new rules in their memories.

D. Problem Definition

In this paper, we consider the problem of updating the forwarding behaviour of all switches in G in the minimum number of rounds, while guaranteeing LF and WPE. We call this problem P_{round} which is proven to be NP-hard [9]. A closely related problem, denoted by P_{switch}, is the problem of maximizing the number of updated switches in a single round, which is NP-hard as well [23]. Although one can solve P_{round} by solving P_{switch} in successive rounds, this approach may encounter a deadlock and consequently fail to update a problem instance that is actually feasible [5].

III. Heuristic Rule Update Algorithm

In this section, we design a fast heuristic algorithm, called Reduced Round Scheduler (RRS), RRS utilizes the structure of the problem to obtain two different update schedules by successively solving the P_{switch} problem and then reduces the required number of rounds by merging those schedules.

A. Fast State Generation

Let $\mathcal{I}$ denote the set of switches, for which updating any single switch $v \in \mathcal{I}$ does not violate the network consistency requirements. Our goal in this sub-section is to compute $\mathcal{I}$.

Label the switches along the *current* path from s to d with natural numbers in an increasing order. These numbers show the order of switches that packets visit on their path from s to d. Let $L{:}V \rightarrow \mathbb{N}$ denote the labeling function and consider two switches u and v. Assume that after the update, u connects to v. If $L(v) < L(u)$, RLF is violated because u sends packets back to an already visited switch v. Likewise, if $L(u) < L(wp) < L(v)$, WPE is violated because a packet that reaches v never goes back to visit wp. Thus, updating switch u (that afterward connects to switch v) is consistent (*i.e.*, $u \in \mathcal{I}$) if and only if one of the following conditions hold:

$$L(wp) \leq L(u) < L(v) \quad (1)$$

$$L(u) < L(v) \leq L(wp) \quad (2)$$

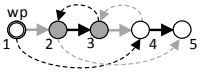


Fig. 3: Unreachable switches (filled gray) can form a loop, not having a reachable successor. In this situation, L is 0 and a reachable switch (*e.g.*, switch 4) that connects to them can not be updated.

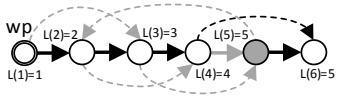


Fig. 4: Label of the unreachable switch 5 (filled gray) is 5. Therefore, switch 3 can update and connect to it consistently. Also, updating switch 5 is consistent because it is unreachable. But updating both switches creates a loop.

Clearly we can update any unreachable switch without violating RLF or WPE. However, after updating u, switch v (u's next switch) may be unreachable. Thus, we extend the definition of L to unreachable switches in order to consistently apply conditions (1) and (2). To this end, we define a *successor relation* on the set of switches and denote it with function $S{:}V \rightarrow V$. Let $S(u)$ denote the current next hop of u which is determined from π_{old} if u is not updated, and from π_{new}, otherwise. The descendants of switch u are all those switches that are reachable from u. The label of an unreachable switch is equal to the minimum label of its descendants. With this definition, we can compare the label of switch u with the first reachable descendant of v to ensure RLF and WPE are satisfied based on conditions (1) and (2). If an unreachable switch has no reachable descendant (see Fig. 3) the label of that unreachable switch is defined as 0. By this definition, reachable switches can not be updated to send packets to such unreachable switches. With the extended definition, L is now defined for all switches in V. Thus, we can compute $\mathcal{I}$.

Theorem 1. *$\mathcal{I}$ can be computed in $O(|V|)$.*

Proof. The induced graph that represents the problem has $|V|$ nodes and $2(|V|-1)$ edges (π_{old} and π_{new}). Therefore we can use Breadth-First search algorithm to compute function L in $O(|V|)$. We can test conditions (1) and (2) in $O(1)$, therefore $\mathcal{I}$ can be computed in $O(|V|)$. □

We still can not update the switches in $\mathcal{I}$ altogether. Lemma 1 presents a negative result about $\mathcal{I}$.

Lemma 1. *Updating a subset of $\mathcal{I}$ may cause RLF or WPE violation.*

Algorithm 1 SWITCHX

Input: G(V, E). ▷ Graph described in Sect. II
Output: $\mathcal{U}, \mathcal{R}$.

```
1: procedure SWITCHX
2:     L(G)                ▷ Label switches with function L described in Sect. III-A
3:     for (u, v) ∈ π_new do
4:         if u.updated == False then
5:             if G.wp.L ≤ u.L ≤ v.L or u.L ≤ u.L ≤ G.wp.L then
6:                 I.add(u)
7:     for v ∈ I do
8:         if v is reachable from s then R.add(v)
9:         else U.add(u)
10:    return U, R
```

Proof. Consider Fig. 4 in which updating switch 3 is consistent because $3 = L(3) < L(5) = 5$. Updating switch 5 is also consistent because it is unreachable. However, updating these two switches simultaneously creates a loop $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2)$. There exists a similar scenario for violation of WPE. □

Our goal is to update the maximum number of switches in $\mathcal{I}$, while preserving RLF and WPE. Observe that if we only update the reachable switches, denoted by $\mathcal{R}$, or only unreachable switches, denoted by $\mathcal{U}$, the update is consistent. A switch is unreachable if packets cannot reach it by following the current forwarding rules starting from the source switch s. We can determine the reachability of switches while computing $\mathcal{I}$ without increasing the complexity of the algorithm. We call the algorithm that uses the function L to update the switches in a single round SWITCHX. SWITCHX is outlined in Algorithm 1. Lemma 2 proves that either $\mathcal{R}$ or $\mathcal{U}$ contains at least half of the maximum number of switches that can be consistently updated in a single round.

Lemma 2. *Updating switches in the bigger set of $\mathcal{R}$ and $\mathcal{U}$ yields a 2-approximation algorithm for the problem P_{switch}.*

Proof. Updating any switch in $\mathcal{I}$ individually is consistent. Therefore, the optimum solution is smaller than $\mathcal{I}$. Furthermore, since every switch that can be updated consistently is either reachable (a member of $\mathcal{R}$) or unreachable (a member of $\mathcal{U}$), we have $\mathcal{I} = \mathcal{R} \cup \mathcal{U}$. Now assume, for the sake of contradiction, that $|\mathcal{R}| < \frac{1}{2}|\mathcal{I}|$ and $|\mathcal{U}| < \frac{1}{2}|\mathcal{I}|$. Since $\mathcal{R} \cap \mathcal{U} = \phi$, we have,

$$|\mathcal{R} \cup \mathcal{U}| = |\mathcal{R}| + |\mathcal{U}| - |\mathcal{R} \cap \mathcal{U}| = |\mathcal{R}| + |\mathcal{U}| < \frac{1}{2}|\mathcal{I}| + \frac{1}{2}|\mathcal{I}| = |\mathcal{I}|$$

However, this contradicts the fact that $\mathcal{I} = \mathcal{R} \cup \mathcal{U}$. □

B. Deadlock-Free Update Scheduler

Our goal is to solve the problem P_{round} by applying SWITCHX iteratively. Since SWITCHX is a linear-time 2-approximation algorithm, the iterative solution is expected to result in a reasonably efficient algorithm. However, it may result in a deadlock [5]. To avoid deadlocks, first we plug SWITCHX into a search algorithm, called Deadlock-Free Update (DFU), and then apply a MERGE algorithm to reduce the number of rounds, as described in the following.

DFU uses a list of $|V|$ booleans to represent the state of the switches, *i.e.*, whether a switch is updated or not. DFU starts from the state where no switch is updated. Then, to search the solution space (which has at most $2^{|V|-1}$ states), it applies

Algorithm 2 DFU: Deadlock-Free Update Scheduler

Input: G(V, E). ▷ Graph described in Sect. II
Output: A deadlock-free update schedule.

```
1: procedure DFU
2:     init_conf ← [false] * |V|
3:     q ← priority_queue(init_conf)
4:     while q.has_next do
5:         cur_conf ← q.head
6:         L(G, cur_conf)                ▷ L is described in Sect. III-A
7:         U, R ← SWITCHX(G)
8:         for candid ⊂ U or R do
9:             next_conf ← UPDATE(cur_conf, candid)          ▷ See Sect. III-B
10:            if next_conf not visited then
11:                if false ∉ next_conf then return solution
12:                else q.add(next_conf)
13:    return failure
```

SWITCHX to *incrementally* generate the next consistently reachable states. DFU uses a priority queue to explore the states with higher number of updated switches sooner, and accelerates the discovery of the final state. The path that connects the initial state to the final state is a consistent update schedule of all switches.

DFU is outlined in Algorithm 2. In line 2, a list of $|V|$ false entries is used to show the initial state. The priority queue is initialized in line 3. In line 6, the function L is computed for all switches. Then, SWITCHX is called in line 7, to calculate sets $\mathcal{R}$ and $\mathcal{U}$. In lines 8 to 12, DFU generates the new states and checks if the final state has been found. Specifically, the UPDATE method takes the current state and a set of switches denoted by `candid` whose update is consistent, and generates another state which is identical to the input state except for the switches in `candid` that are updated to their final states. In favor of brevity and readability, details of bookkeeping to retrieve the solution (line 11) and ensuring that each state is visited once (line 10) are omitted.

C. RRS Algorithm

The Reduced Round Schedule (RRS) algorithm employs DFU to find two different schedules for the update problem and then merges the schedules to build another update schedule that has fewer number of rounds. We demonstrate the intuition behind the algorithm through an example.

Example: Consider the network in Fig. 5(a), and let switch 6 be the waypoint. DFU updates the maximum number of switches (i.e. $\{1, 2\}$) in the first round, while in the next 4 rounds only 1 switch can be updated consistently. In the second round, only switch 3 can be updated because updating switches 4 and 6 creates the loops $(4 \rightarrow 3 \rightarrow \dots)$ and $(6 \rightarrow 5 \rightarrow \dots)$, respectively, and updating switch 5 violates WPE. In the third round, we can only update switch 4. Finally, switches 5 and 6 are updated in the fourth and fifth rounds, respectively. Therefore, it finds an update schedule with 5 rounds. Now, create a new problem instance by swapping π_{old} and π_{new}. The result is depicted in Fig. 5(b). DFU updates this network in 3 rounds ($\{1, 6\}$ in the first round, then $\{3, 4, 5\}$, and finally $\{2\}$). Note that, the second schedule updates the first problem in 3 rounds if we apply it in the reverse order.

Next, we prove that if we swap π_{old} and π_{new} in a network, the solution of the new problem (updating π_{new} to π_{old}) correctly solves the original problem (updating π_{old} to π_{new})

(a) Original Problem (b) Transformed Problem

Fig. 5: An example in which DFU finds a better solution if we transform the problem by swapping π_{old} and π_{new}.

if applied in reverse order of rounds. We refer to this property as *reversibility*.

Theorem 2. *The problem of consistent network update with RLF and WPE properties is reversible.*

Proof. Assume that in a problem instance all switches are updated. Let s_f denote the final state of the network (*i.e.*, state of all switches). Suppose that reverting the update of a maximal set w of switches is consistent and takes the network to another state s_{f-1}. By the definition of consistency, any state that is obtained by reverting the update of any subset $w' \subseteq w$ is also consistent. Furthermore, each state that is obtained from s_f by reverting the update of the switches in $w' \subseteq w$, is exactly the state that is obtained from s_{f-1} by updating the switches in $w - w'$. Since any state that is reachable from s_f by reverting the update of switches in w is consistent, all the states reachable from s_{f-1} by applying w are also consistent. Since in each round the initial and final states are consistent, by induction, it can be shown that a consistent schedule for the reversed version of the problem is also a consistent schedule for the original problem. □

The RRS algorithm is presented in Algorithm 3. It uses DFU to solve the original and reverse problems and then merges the solutions to reduce the number of rounds. We refer to the solutions of the original and reverse problem as *forward* and *backward* schedules, respectively. The MERGE algorithm which is used to combine the forward and backward schedules is described next.

MERGE Algorithm: Let $w_{i,r} \subset V$ be the set of switches that schedule $i \in \{1, 2\}$ (where, $i = 1$ refers to the *forward* and $i = 2$ refers to the *backward* schedule) has updated up to (and including) round r. Denote the maximum round number in the schedule i by R_i. For each schedule i, create a directed graph $G_i(V_i, E_i)$ as follows. For each round r of the schedule i, add a node to graph G_i. Each node stores three values r, $w_{i,r}$, and $R_i - r$. Note that $R_i - r$ shows the required number of rounds to complete schedule i starting from round r. In each graph, add the directed edge $v_{\{r,..\}}$ to $v_{\{r+1,..\}}$ (“.” means wildcard) to show a transition from round r to $r + 1$ in the corresponding schedule. Consider a pair of nodes $v_{\{.,w_{i,r},d\}}, v'_{\{.,w'_{i',r'},d'\}}$ which are not in the same graph. If $w_{i,r} \subset w'_{i',r'}$ and $d' < d-1$, and we can consistently update the switches in $w'_{i',r'} - w_{i,r}$ in a single round, then add a directed edge from v to v'. Such links build schedules which are better than schedule i by at least one round. We can reduce the time of merging the graphs by observing that, if the transition from node $v_{\{.,.,d\}}$ to $v'_{\{.,.,d'\}}$ is not consistent, the transition to any other node $v''_{\{.,.,d'<d''\}}$ is not consistent either. Finally, add two nodes S and D. Connect the source and destination of the graphs G_i to S and D with proper directions, respectively. Clearly, the length of the shortest path from S to D minus two is the length of a consistent update schedule which is never longer than any of the input schedules.

Algorithm 3 RRS: Reduced Round Scheduler

Input: $G_1(V_1, E_1)$. ▷ Graph described in Sect. II
Output: A Scheduling if exists.

```
1: procedure RRS(G_1(V_1, E_1))
2:     G_2(V_2, E_2) ← G_1.copy()
3:     for e ∈ E_2 do
4:         if e.label == π_old then e.label ← π_new
5:         else e.label ← π_old
6:     sol_fwd ← DFU(G_1)
7:     sol_bwd ← DFU(G_2)
8:     return MERGE(sol_fwd, sol_bwd)        ▷ Defined in Sect. III-C
```

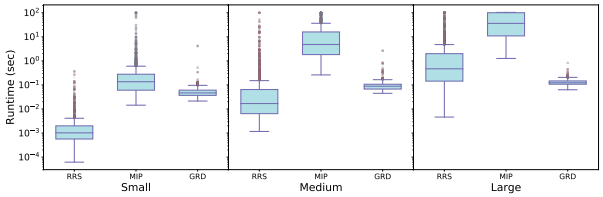


Fig. 6: Runtime comparison of RRS, MIP, and GRD. Y-axis is logarithmically scaled. The red marks are outliers that are $1.5 \times IQR$ (Interquartile Range) or more above (below) the third (first) quartile.

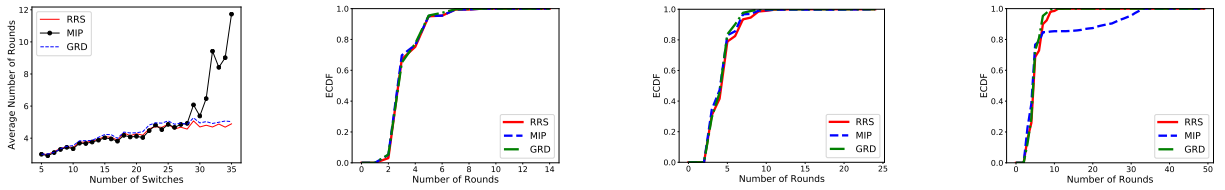
IV. PERFORMANCE EVALUATION

Setup and Parameters. We evaluate RRS by comparing it with the exact solution of Problem MIP (denoted by MIP) and the greedy algorithm proposed in [23] (denoted by GRD) which solves P_{switch} optimally. All algorithms terminate after 100 seconds if they don't find a solution or show infeasibility. Note that, if MIP can not find the optimal solution before 100 seconds, it will return the best found feasible solution. Therefore, in practice other algorithms may perform better than MIP. We implemented all algorithms in Python and used Gurobi 8.0 to solve the optimization problems. We conducted the experiments on a machine with Intel(R) 2.10 GHz Xeon(R) CPU, 16GB memory, and Ubuntu 16.04 as the operating system. We also present the results of running DFU on the original and reverse problem instances (Sect. III-C) and denote them by FWD and BWD, respectively.

Data Set. We use a public dataset of $28,581$ scenarios[1]. For each problem instance, the new path, π_{new}, and the only waypoint, wp, are generated randomly. The number of switches in a path ranges from 5 to 35. Since the length of any path is less than or equal to the diameter of the network, the network instances considered in our experiments cover a wide range of network sizes.

Runtime Comparison. The scenarios are divided into three groups: (1) Small scenarios with path lengths $\leq$15, (2) Medium scenarios with path lengths 15 to 25, and (3) Large scenarios with path lengths >25. There are approximately 9K problem instances in each group. Figure 6 shows the runtime distribution of RRS, MIP, and GRD algorithms. Even in the small scenarios, the superiority of RRS is evident.

[1]The dataset is publicly available at http://net.t-labs.tu-berlin.de/∼stefan/netup.tar.gz



(a) Average # of Rounds (b) Small scenarios (c) Medium scenarios (d) Large scenarios

Fig. 7: Average and empirical CDF of the number of rounds for different number of switches.

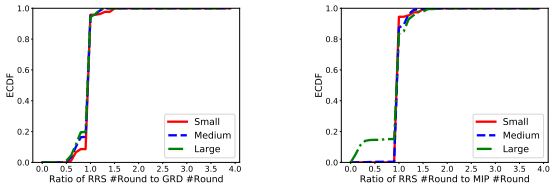
While RRS solves all small scenarios in less than 1 second, in some scenarios MIP reaches the 100 seconds time limit (note the logarithmic y-axis). Runtime medians of running RRS on small, medium and large scenarios are 9×10^{-4}, 0.014, and 0.46 seconds, respectively, which compared to the ones obtained by MIP, namely 0.12, 4.7, and 36 seconds, show 323x speedup. The runtime medians of running GRD on small, medium and large scenarios are 0.047, 0.089, and 0.12 seconds, respectively, in which RRS shows 5.5x speedup in comparison. The efficiency of SWITCHX is the main reason behind the performance of RRS.

Ability to Find a Solution. Table I summarizes the ratio of solved, infeasible, and failed instances. GRD shows its limitation even on small scenarios. Specifically, it fails to solve 1250 small instances (*i.e.*, 13%) which are solved by other algorithms. Likewise, MIP fails to solve 43 small instances (*i.e.*, 0.4%) which are solved by RRS. Note that, for small and medium scenarios, there is actually no room for improvement, and thus the performances of the algorithms resemble each other. However, in large scenarios, RRS outperforms MIP and GRD, and achieves near optimal performance. In general, RRS solves 6% and 14.7% more instances compared to MIP and GRD. Furthermore, RRS solves 109 more instances compared to the FWD and BWD algorithms.

Number of Rounds Comparison. Average number of rounds obtained by different algorithms is represented in Fig. 7(a). For small scenarios, the performances of all algorithms are similar. Most notably, the average number of rounds under RRS is always lower than that of other algorithms. We can see that MIP is not scalable, and for large scenarios, its performance degrades significantly. Figs. 7(b), 7(c), and 7(d) show the empirical cumulative distribution functions (ECDF) of the number of rounds achieved by the algorithms in different scenarios. We observe that the majority of small scenarios are solved in at most 5 rounds, while 10 rounds are needed to solve almost all medium and large scenarios. However, in large scenarios, MIP shows a considerably lower performance and sometimes computes a schedule with more than 30 rounds, while RRS never uses more than 15 rounds. Note that, GRD fails to solve the complicated problem instances (*e.g.*, it fails to solve 1700 large scenarios). Therefore, Fig. 7(d) can only capture the performance of GRD on easier problem instances. Consequently, Fig. 7(d) shows that GRD schedules rule updates in a small number of rounds. RRS, however, solves more problem instances, and at the same time finds schedules that have fewer number of rounds. This means that RRS computes good schedules even for complicated problem instances.

TABLE I: Ratio of solved, infeasible and failed instances

Size	Status	Algorithm				
		RRS	MIP	GRD	FWD	BWD
Small	Solved	0.90	0.90	0.86	0.90	0.90
	Infeasible	0.09	0.08	0.0	0.09	0.09
	Failure	0.0	0.004	0.13	0.0	0.0
Medium	Solved	0.95	0.94	0.82	0.95	0.95
	Infeasible	0.04	0.006	0.0	0.04	0.04
	Failure	10^{-4}	0.05	0.18	0.001	10^{-4}
Large	Solved	0.99	0.84	0.80	0.98	0.98
	Infeasible	0.002	10^{-4}	0.0	0.001	0.001
	Failure	0.006	0.15	0.19	0.01	0.01



(a) RRS vs. GRD (b) RRS vs. MIP

Fig. 8: Empirical CDF for the ratio of the number of rounds of different algorithms.

Effect of MERGE algorithm. Consider the ECDF of the ratio of the number of rounds among different algorithms in Fig. 8. Since RRS uses the approximate SWITCHX algorithm we should expect an inferior performance compared to GRD which uses the optimal algorithm for P_{switch} problem. However, Fig. 8(a) shows that RRS, by using the MERGE algorithm and combining the solutions of BWD and FWD, achieves 10% to 20% improvement compared to GRD for different scenario sizes. Figure 8(b) compares RRS and MIP. We can see that RRS reduces the number of rounds in 15% of large scenarios. Furthermore, there are scenarios in which the ratio of the number of rounds between RRS and MIP is close to zero. This means that MIP is not scalable, and under the same time constraint as RRS, it performs considerably worse.

V. CONCLUSION

We designed RRS algorithm to update SDNs with minimum number of rounds under RLF and WPE. RRS uses three novel building blocks and significantly reduces the time complexity of solving the problem. Extensive evaluations showed that our algorithm is able to find schedules that are efficient in terms of the number of rounds. In the future, we plan to consider other types of consistency properties.

REFERENCES

[1] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, 2013.

[2] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," *Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.

[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, 2012, pp. 323–334.

[4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," *Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, 2013.

[5] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *ACM HotNets*, 2014, pp. 15:1–15:7.

[6] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *ACM SIGCOMM*, 2012, pp. 13–24.

[7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions," in *CNSM*, 2015, pp. 50–56.

[8] L. J. Chaves, I. C. Garcia, and E. R. M. Madeira, "An adaptive mechanism for LTE P-GW virtualization using SDN and NFV," in *CNSM*, 2017, pp. 1–9.

[9] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," *Perform. Eval. Rev.*, vol. 44, no. 1, pp. 273–284, 2016.

[10] L. Qu, C. Assi, K. Shaban, and M. Khabbaz, "A reliability-aware network service chain provisioning with delay guarantees in nfv-enabled enterprise datacenter networks," *IEEE Trans. Netw. Service Manag.*, vol. PP, no. 99, pp. 1–1, 2017.

[11] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid, "Efficient loop-free rerouting of multiple sdn flows," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 948–961, 2018.

[12] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *ACM HotNets*, 2013, pp. 20:1–20:7.

[13] K. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent network updates," *CoRR*, vol. abs/1609.02305, 2016. [Online]. Available: http://arxiv.org/abs/1609.02305

[14] J. Moy, P. Pillay-Esnault, and A. Lindem, "Graceful OSPF restart," RFC 3623, 2003.

[15] A. Shaikh, R. Dube, and A. Varma, "Avoiding instability during graceful shutdown of multiple ospf routers," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 532–542, 2006.

[16] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1280–1292, 2007.

[17] S. Brandt, K. T. Förster, and R. Wattenhofer, "On consistent migration of flows in SDNs," in *IEEE INFOCOM*, 2016, pp. 1–9.

[18] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *ICNP*, 2015, pp. 1–10.

[19] H. Xu, Z. Yu, X. Y. Li, L. Huang, C. Qian, and T. Jung, "Joint route selection and update scheduling for low-latency update in SDNs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, 2017.

[20] J. Zheng, H. Xu, G. Chen, H. Dai, and J. Wu, "Congestion-minimizing network update in data centers," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[21] K. T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *IFIP Networking*, 2016, pp. 1–9.

[22] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM HotSDN*, 2013, pp. 49–54.

[23] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently consistent sdn updates: Being greedy is hard," in *Structural Information and Communication Complexity*, 2016, pp. 391–406.

[24] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *ACM PODC*, 2015, pp. 13–22.

[25] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *ACM SIGPLAN*, 2015, pp. 196–207.

[26] S. Vissicchio and L. Cittadini, "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates," in *IEEE INFOCOM*, 2016, pp. 1–9.