Transiently Policy-Compliant Network Updates

Arne Ludwig[1] Szymon Dudycz[2] Matthias Rost[1] Stefan Schmid[3]
[1] TU Berlin, Germany [2] University of Wroclaw, Poland [3] University of Vienna, Austria

***Abstract*—Computer networks have become a critical infrastructure. It is hence increasingly important to guarantee a correct, consistent and secure network operation at any time, even during route updates. However, most existing works on consistent network update protocols focus on connectivity properties only (e.g., loop-freedom) while ignoring basic (security) policies.**

This paper studies how to update routes in a software-defined network in a transiently policy-compliant manner. In particular, our goal is to enforce waypoints: at no point in time should it be possible for packets to bypass security critical network functions (such as a firewall). This problem is timely, given the advent of network function virtualization which envisions more flexible middlebox deployments, not limited to the network edge.

This paper shows that enforcing waypoint traversal in transient states can be challenging: waypoint enforcement can conflict with loop-freedom. Even worse, we rigorously prove that deciding whether a waypoint enforcing, loop-free network update schedule exists is NP-hard. These results hold for both kinds of loop-freedom used in the literature: strong and relaxed loop-freedom. This paper also presents optimized, exact mixed integer programs to decide feasibility quickly and to compute optimal update schedules. We report on extensive simulation results, and also study scenarios where entire "service chains", connecting multiple waypoints, need to be updated consistently.

I. Introduction

Computer networks are becoming more and more programmable and flexible. In particular, the software-defined networking paradigm enables a logically centralized operation of computer networks: in a Software-Defined Network (SDN), a software controller can install, update and verify "the paths that packets follow", i.e., the (routing) *policies* [7], fast and in a globally consistent manner.

However, today, we do not yet have a good understanding of the opportunities and limitations of a more dynamic network management in general, and the Software-Defined Network (SDN) paradigm in particular. Over the last years, especially the problem of consistent network updates has received much attention [12], [18], [20], [27], [30]. While the logically centralized control introduced by software-defined networking is appealing, an SDN still needs to be regarded as a distributed system, posing non-trivial challenges: in particular, the communication channel between switches and controller exhibits non-negligible and varying delays [9], [20], [23]. These delays may result in temporary misconfigurations, i.e. invalid network configurations, diverging from the expected control plane view [22].

In a first line of works, initiated by Reitblatt et al. [30], network updates providing strong consistency guarantees have been studied: even during the transition from an old routing policy π_1 to a new routing policy π_2, the *Per-Packet Consistency* (PPC) property is ensured, i.e., each packet will either be forwarded according to π_1 (exclusively-) or π_2, but not a combination of both. In a second line of works, initiated by Mahajan and Wattenhofer [27], weaker transient consistency properties have been investigated: during a network update, a packet may be forwarded according to the old policy π_1 at some switches and according to the new policy π_2 at other switches; however, the update still provides basic *transient guarantees*, such as *Loop-Freedom* (LF): packets will never be forwarded along a loop. Providing only *weak* transient guarantees, like loop-freedom, is an attractive alternative to stronger consistency models such as PPC: parts of the updates can take effect earlier, update protocols can be more resource efficient (e.g., there is no need for extra rules on the switch), and tagging is not required (tagging is often problematic given the limited packet header space). In fact, Ludwig et al. [12] have observed that there exist multiple consistency levels even for loop-freedom itself: besides the canonical *strong* loop-freedom (studied in [27]), a *relaxed* notion of loop-freedom may facilitate even faster network updates.

However, ensuring connectivity properties such as loop-freedom alone is often insufficient in practice: especially in security critical environments, additional consistency guarantees are required, related to the network's (security) policies. A particularly important policy is *Waypoint Enforcement* (WPE): the traversal of specific network functions or middleboxes must be ensured for each packet. In fact, today's computer networks consist of a large number of so-called *middleboxes*, providing a wide spectrum of in-network functionality for security, performance, policy compliance, etc. For example, network policies are often defined in terms of adjacency matrices or big switch abstractions, specifying which traffic is allowed between an ingress network port s and an outgress network port d [21]. In order to enforce such a policy, traffic from s to d needs to traverse a middlebox instance inspecting and classifying the flows. In fact, the number of middleboxes in enterprise networks can be of the same order of magnitude as the number of routers [19]. Middleboxes can also be virtualized [1] and hence be deployed fast and flexibly e.g., as a virtual machine on a commodity x86 server. Along with network function virtualization also comes the trend to deploy middleboxes not only at the edge, but also in the network core, potentially reducing deployment costs but also rendering waypoint enforcement more challenging.

A. Contributions

This paper studies an important class of network update problems: updates which respect the (security) policy of the network, by enforcing waypoints. In a nutshell, a weakly consistent flow rerouting algorithm providing waypoint enforcement guarantees that *before*, *during* and *after* the transition from route π_1 to route π_2, packets traverse a certain middlebox (the waypoint). Prior work on network updates mainly focused on basic forwarding properties related to connectivity, most prominently loop-freedom [13].

We make the following contributions:

1) **Waypoint enforcement matters:** We show that waypoints may easily be bypassed during a route update, if no countermeasures are in place. Moreover, we prove that Loop-Freedom (LF) and Waypoint Enforcement (WPE) cannot always be implemented in a wait-free manner, in the sense that the controller must rely on an upper bound estimation for the maximal packet latency in the network.
2) **LF and WPE may conflict:** We observe that the transient Waypoint Enforcement property (WPE) may conflict with Loop-Freedom (LF), in the sense that ensuring both properties may not be possible simultaneously. We also prove that relaxing the notion of loop-freedom, as suggested by Ludwig et al. [12] for performance reasons, does not help to render impossible instances feasible: a problem instance which cannot be solved under strong loop-freedom and waypoint enforcement, cannot be solved under relaxed loop-freedom and waypoint enforcement either.
3) **NP-hardness:** The main technical result of this paper is a formal proof that the decision problem whether both consistency properties, LF and WPE, can be satisfied simultaneously is NP-hard. This result holds for both strong and relaxed loop-freedom.
4) **Multiple waypoints:** We initiate the discussion of how to consistently update routes with more than one waypoint, i.e. *service chains*. In particular, we show that flexibilities in the order in which service chain functions are traversed, cannot be utilized to render the updating problem easier.
5) **Algorithms:** We present efficient algorithms for many network update variants, as well as optimized, exact mixed integer programming algorithms for the general problem.
6) **Simulations:** We report on an extensive simulation study. In particular, we find that the number of nodes as well as the number of waypoints significantly impacts the runtime. While computing the schedule with a minimum number of rounds is the main objective, we show that for many scenarios the feasibility or the infeasibility can be decided quickly.

B. Paper Organization

The remainder of this paper is organized as follows. Section II introduces our formal model. Section III presents first intuitions and insights. The formal NP-hardness proof is given in Section IV. Section V initiates the discussion of multiple waypoints. Section VI presents mixed integer programs for solving the network update problem together with several optimizations. Simulation results are discussed in Section VII. After reviewing related work in Section VIII, we conclude the paper in Section IX.

II. Formal Model

We consider a Software-Defined Network (SDN) which is managed by a controller, communicating flow rules to the switches across an asynchronous communication channel. As the updates occur asynchronously, we require the controller to send out updates only to a "safe" subset of nodes at any time. Only after these updates have been confirmed (using acknowledgments), updates for the next subset can be issued.

A. Representation of Policy Updates

The controller needs to change an old policy resp. *route* π_1 to a new policy resp. *route* π_2. Both π_1 and π_2 follow simple directed paths. Initially, packets are forwarded along π_1 (henceforth also called *old edges*), and eventually they should be forwarded according to the new rules of π_2. Packets should never be delayed or dropped at a switch, henceforth also called *node*: whenever a packet arrives at a node, a matching forwarding rule should be present.

Without loss of generality, we assume that π_1 and π_2 lead from a source s to a destination d. Since nodes appearing only in one of the two paths are trivially updateable, we focus on the network G induced by the nodes V which are part of *both* policies π_1 *and* π_2, i.e., $V = \{v : v \in \pi_1 \wedge v \in \pi_2\}$. Thus, we can represent the policies as paths $\pi_1 = \langle s = v_0, v_1, \ldots, v_{\ell-1} = d\rangle$ and $\pi_2 = \langle s = v_0, \pi(v_1), \ldots, \pi(v_{\ell-2}), v_{\ell-1} = d\rangle$, for some permutation $\pi : V \setminus \{s, d\} \to V \setminus \{s, d\}$ and some number ℓ. Additionally, edges (v_i, v_{i+1}) being contained in both π_1 and π_2 can be removed as the node v_i does not need to be updated in the first place. In fact, the edges lying on the old and the new policy can be removed by contracting the nodes of the respective edges to become a single node. Hence, we assume that edges in π_1 and π_2 are *disjoint*.

Figure 1 illustrates the model [12]: depicted on the left is an old policy π_1 (*solid*) and a new policy π_2 (*dashed*). As discussed above, only the updateable nodes which are shared by the two policies are of interest. Thus, the update problem can be reduced to the 5-node chain graph (*right*).

In the following we call an edge (u, v) *of the new policy* π_2 *forward*, if v is closer to the destination, resp. *backward*, if u is closer to the destination (both times with respect to π_2). Similarly, we refer to nodes having a forward edge as *forward nodes* and to nodes having a backward edge as *backward*

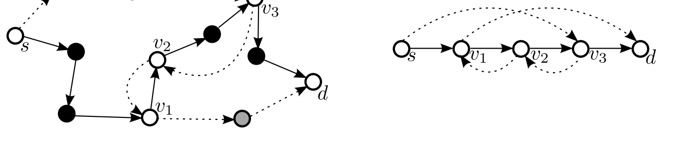


Fig. 1. Overview of model and reduction. The solid lines show the old policy π_1 and the dashed lines show the new policy π_2. The update problem on the *left* boils down to the update problem for the line representation depicted on the *right* (the old route goes from left to right). Nodes shown in white are the only ones which are part on both paths, and hence relevant for the problem.

nodes. Hence, the edges (s, v_3) and (v_3, v_2) of the new policy depicted in Figure 1 (*left*) are forward and backward edges, respectively, and node s is a forward node while v_3 is a backward node.

B. The Network Update Problem

The task is to find an update schedule $U = \langle U_1, U_2, \ldots, U_k \rangle$, i.e. a sequence of switches $U_t \subseteq V$ to be updated "asynchronously" in rounds $t = 1, \ldots, k$, such that after round k the new policy π_2 is in effect. Note that the updates of round t are only issued by the controller if the updates of round $t-1$ have been acknowledged by the switches contained in U_{t-1}. As any node $v \in V \setminus \{d\}$ is updated exactly once, the subsets U_t form a partition of the nodes: $V \setminus \{d\} = \bigsqcup_{t=1,\ldots,k} U_t$.

The objective is to find update schedules ensuring properties as loop-freedom and waypoint enforcement even in transient states (see below) while minimizing the number of update rounds k. The reduction of the number of rounds is a natural objective, given the time it takes to update an individual OpenFlow switch today [9], [20], [23].

C. Transient Forwarding States and Graphs

We introduce the following notation to formally capture the transient forwarding states possible when asynchronously updating the switches within a round. Let $U_{<t} = \bigcup_{i=1,\ldots,t-1} U_i$ denote the set of nodes which have already been updated before round t, and let $U_{\leq t}$, $U_{>t}$ etc. be defined analogously. Furthermore, let $\delta_1^+(v)$ and $\delta_2^+(v)$ denote the outgoing edge of node $v \in V$ according to policy π_1 and π_2, respectively. Similarly, we denote by $out_1(v)$ and $out_2(v)$ the outgoing neighbors of v, i.e. the heads of $\delta_1^+(v)$ and $\delta_2^+(v)$, respectively. Moreover, we extend the definition of δ_i^+ to entire node sets $S \subseteq V$, i.e., $\delta_i^+(S) = \{\delta_i^+(v) \mid v \in S\}$, for $i \in \{1, 2\}$.

Since updates in round t are performed asynchronously, any subset $X \subseteq U_t$ of nodes might be updated, while the nodes $\overline{X} = U_t \setminus X$ are not updated. To capture the forwarding behavior in round t given that the nodes $X \subseteq U_t$ are updated, we introduce the *transient forwarding graph* $G_t^X = (V, E_t^X)$ on the set of nodes V. As any node $v \in U_{<t} \cup X$ is updated, the forwarding on v is performed according to the new policy π_2. On the other hand, the nodes $U_{>t} \cup \overline{X}$ are not updated and hence for these nodes the old policy π_1 is in effect. Hence, $E_t^X = \delta_1^+(U_{>t} \cup \overline{X}) \cup \delta_2^+(U_{<t} \cup X)$ holds.

Lastly, note that by setting $X = U_t$, the respective transient forwarding graph G_t^X also describes the forwarding state *after* round t. Hence, when enforcing a property for each round t and each subset $X \subseteq U_t$, the respective property will be enforced throughout the update process.

D. Transient Forwarding Properties

Following [12], we consider two types of Loop-Freedom (LF), namely *Strong Loop-Freedom* (SLF) and *Relaxed Loop-Freedom* (RLF) and introduce the notion of waypoint enforcement (WPE). Throughout the paper, we use the term LF whenever a result holds both for SLF and RLF. Furthermore, if both WPE and RLF or SLF are enforced we simply write RLF + WPE and SLF + WPE, respectively.

*1) Strong Loop-Freedom (*SLF*):* Strong Loop-Freedom requires that the update schedule $U = \langle U_1, U_2, \ldots, U_k \rangle$ fulfills the property that for all times t and for any subset of updated nodes $X \subseteq U_t$, the transient forwarding graph G_t^X is loop-free. In other words, all intermediate forwarding topologies form DAGs.

*2) Relaxed Loop-Freedom (*RLF*):* While strong loop-freedom forbids the existence of *any* loop in any of the transient forwarding graphs G_t^X, relaxed loop-freedom only forbids loops reachable from the source s: for any round t and for any subset X the graph G_t^X must not contain a loop *reachable from the source s*.

Relaxed Loop-Freedom is motivated by the practical observation that transient loops are only harmful as long as the loop is connected to the source and hence arbitrarily many packets may loop. Accordingly, if relaxed loop-freedom is enforced, only a constant number of packets can loop: new packets are never pushed into a loop "at line rate". In other words, even if nodes acknowledge new updates late (or never), new packets will not enter loops.

*3) Waypoint Enforcement (*WPE*):* Waypoint Enforcement is the property that each packet must traverse a special node $wp \in V$ of the network, henceforth called the *waypoint*. For instance, the waypoint could be a firewall or intrusion detection system which each packet must traverse even in transient states. WPE is defined as follows: for any round t and any subset $X \subseteq U_t$, the transient forwarding graph G_t^X *must not* contain a path from the source s to the destination d *bypassing* the waypoint wp.

This definition easily generalizes to multiple waypoints: if multiple waypoints are given none of the waypoints may be bypassed in any transient forwarding graph G_t^X.

E. Example

Given these definitions, let us consider an extended example, as illustrated in Figure 2. The old policy π_1 connects four nodes (switches), from left to right (depicted as a *straight, solid line*); the new policy π_2 is shown as a *dashed line*. The second node, v_1 (in *black*), represents the waypoint which needs to be enforced.

How can we update the policy π_1 to π_2? A simple solution is to update all nodes *concurrently*. However, as the controller needs to send these commands over the asynchronous network, they may not arrive simultaneously at the nodes, which can result in inconsistent states. For example, if s is updated before v_1 and v_2 are updated, a transient forwarding path emerges which violates WPE: packets originating at s will be sent to v_2 and from there to the destination d: the waypoint v_1 is *bypassed*.

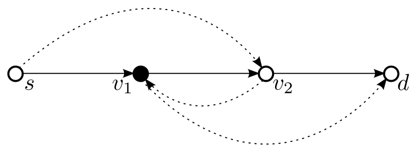


Fig. 2. Examplary network update instance; node v_1 is the waypoint.

One solution to overcome this problem would be to perform the update in *two (communication) rounds*: in the first round, only v_1 and v_2 are updated, and in a second round, once these updates have been performed and acknowledged, the controller also updates s. Note that this 2-round strategy indeed maintains the waypoint at any time during the policy update. However, the resulting solution may still be problematic, as it violates our second desirable transient consistency property, *loop-freedom*: if the update for node v_2 arrives before the update at node v_1, packets may be forwarded in a loop, from nodes v_1 to v_2 and back. Both Waypoint Enforcement WPE as well as Loop-Freedom LF can be ensured (for this specific example) in a *three-round* update: in the first round, only v_1 is updated, in the next round v_2, and finally s. The above example already highlights the complexity of finding the network update schedules using the minimal number of rounds as the LF and WPE property *conflict*. In fact, some network update instances cannot be solved at all (cf. Theorems 3 and 4).

III. First Observations

In this section, we provide some initial insights into the network update problem.

A. Observation 1: You may have to wait

It turns out that the transient enforcement of a waypoint is non-trivial. We first show an interesting negative result: it is not possible to implement WPE in a "wait-free manner", in the following sense: a controller does not only need to wait until the nodes have acknowledged the policy updates of round i before sending out the updates of round $i+1$, but the controller also needs some estimate of the maximal packet latency: if a packet can take an arbitrary amount of time to traverse the network, it is never safe to send out a policy update for certain scenarios. We are not aware of any other transient property for which such a negative result exists. For ease of presentation, we use the notation $\pi_i^{<wp}$ to refer to the first part of the route given by policy π_i, namely the sub-path from the source to the waypoint, and $\pi_i^{>wp}$ to refer to the second part from the waypoint to the destination.

Theorem 1. *In an asynchronous environment, a new policy can never be installed without violation of* WPE*, if a node is part of $\pi_1^{<wp}$ and $\pi_2^{>wp}$.*

Proof. Consider the example in Figure 2 again, but imagine that the waypoint is at node v_2 instead of node v_1. Consider the following update strategy (all other strategies are similar): in the first round, s and v_2 are updated, and in the second round, v_1. This strategy clearly ensures WPE, *if (but only if)* the updates of Round 2 are sent out after packets forwarded according to the rules before Round 1 have left the system. However, if packets can incur arbitrary delays, then there could always be packets left which are still traversing the old (*solid*) path from s to v_1. These packets have not been routed via the waypoint (v_2) so far but will be sent out to d by v_1 in the new path, violating the WPE property. □

Algorithm 1: WayUp

```
1 Input: old policy $\pi_1$, new policy $\pi_2$, threshold $\theta$
2 update nodes of $\pi_2$ which are not in $\pi_1$
3 update nodes of $\pi_1^{>wp}$ with backw. rules in $\pi_2^{<wp}$
4 update remaining nodes of $\pi_2^{<wp}$
5 wait $\theta$
6 update nodes of $\pi_2^{>wp}$
```

B. Observation 2: Waypoint enforcement alone is easy

Fortunately, in practice, packets do not incur arbitrary delays, and Theorem 1 may only be of theoretical interest: it is often safe to provide an update algorithm with some good upper bound θ on the maximal packet latency. The upper bound θ can be seen as a parameter to tune the safety margin: the higher θ, the higher the probability that each packet is actually waypoint enforced.

With these concepts in mind, we now describe our algorithm WayUp which always ensures correct network updates, i.e., updates which consistently implement WPE if the maximal packet transmission time is bounded by θ. We define $v_1 \prec_{\pi_i} v_2$ to express that a node v_1 is visited before v_2 on π_i. The update rule (v_2, v_1) with $v_1 \prec_{\pi_1} v_2$ is a backward rule (or backward edge) with respect to the initial direction of the line.

The round complexity of WayUp is *four*: in the first round, all nodes are updated which were not part of the old policy π_1, and therefore do not have an impact on current packets. In the second round, each node behind the waypoint (i.e., $\pi_1^{>wp}$) which is part of $\pi_2^{<wp}$ and which has a backward rule, is updated. This allows us to update the remaining nodes from $\pi_2^{<wp}$ in the third round, since each packet which is sent "behind" the waypoint will eventually come back, according to the consistency properties of the new policy. After this round, the algorithm will wait θ time to ensure that no packet is on $\pi_1^{<wp}$ anymore. In the fourth round it is possible to update all nodes of $\pi_2^{>wp}$ in one round, because the update cannot interfere with $\pi_2^{<wp}$ anymore, and hence it cannot violate WPE.

Theorem 2. WayUp *takes four rounds and guarantees the* WPE *property at any time.*

Proof. The round complexity follows from the algorithm definition. The transient consistency can be proved line-by-line: Line 2 of Algorithm 1 cannot violate WPE since no packet is crossing any of these nodes. Line 3 does not interfere with $\pi_1^{<wp}$ and therefore each packet will still be sent via $\pi_1^{<wp}$ towards the waypoint. As long as π_2 is consistent, any packet that reaches any node of $\pi_2^{<wp}$ will eventually reach the waypoint during Line 4, since all backward rules are already updated and no rule will bypass them. In Line 6, WPE is already guaranteed, since $\pi_2^{<wp}$ is already in place and θ time has elapsed. □

C. Observation 3: Consistency properties may contradict

While WayUp is good and fast at updating routes in a waypoint enforcing manner, it does not provide loop-freedom.

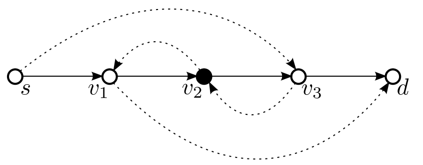


Fig. 3. WPE and LF may conflict.

As we will show next, WPE and LF may even conflict, i.e. it is sometimes impossible to simultaneously enforce both.

Theorem 3. WPE *and* LF *may conflict.*

Proof. Consider the example depicted in Figure 3. Clearly, the source s can only be updated once v_3 is updated, otherwise packets will be sent to d directly, which violates WPE. An update of v_3 can only be scheduled after an update of v_2 without violation of LF. However, v_2 needs to wait for v_1 to be updated for the same reasons. This leaves an update of v_1 as the last possibility, which however violates WPE. Hence no update schedule not violating either WPE or LF exists. □

D. Observation 4: It does not help to relax

Relaxed loop-freedom offers additional flexibility over strong loop-freedom and allows for update schedules of fewer rounds. Given that WPE and LF may conflict, the question arises whether under RLF and WPE more instances can be solved than under SLF and WPE. The following theorem answers this question negatively: all instances solvable under RLF and WPE (henceforth RLF + WPE) are also solvable under SLF and WPE (SLF + WPE).

Theorem 4. *A network update instance is solvable under* SLF + WPE*, if and only if, the instance is solvable under* RLF + WPE.

Proof. As any strong loop-free schedule is also a relaxed one, we only need prove that any instance solvable under RLF + WPE is also solvable under SLF+WPE. For the sake of contradiction, assume that there exists an instance solvable under RLF + WPE which is not solvable under SLF + WPE.

Let $\mathcal{U}$ denote all valid update schedules of the instance under RLF + WPE. For any update schedule $U \in \mathcal{U}$, we denote by $l(U)$ the round in which the first loop (disconnected from the source) is created by U. If U contains no loops at all, we set $l(U) = \infty$. Clearly, as the instance is not solvable under SLF + WPE by assumption, any feasible update schedule must contain a loop at some point in time and hence $l(U)$ is finite. We now pick an update schedule $\hat{U} \in \mathcal{U}$ maximizing $l(\hat{U})$, i.e. an update schedule maximally delaying the creation of the first loop. We denote the round in which the first loop is created by $r = l(\hat{U})$.

Note that we can assume without loss of generality that in any round of $\hat{U}$ only a single update takes place: if multiple updates take place in a round, then these parallel updates can be executed sequentially by introducing additional rounds (one per update) while not decreasing $l(\hat{U})$ and neither violating RLF nor WPE.

Now, let $x \in V$ be the (single) node updated in round $r = l(\hat{U})$, such that the update of x creates a loop (not reachable from the source). We denote by X_r the set of nodes reaching x after all updates of round $r-1$ were successfully installed. As the update of node x in round r creates a loop, any node contained in the set X_r leads (via x) to a loop. Furthermore, any node in X_r is connected to some other node in X_r, which equivalently implies the existence of a loop. Hence, none of the nodes in X_r can be reachable from the source in round r as otherwise RLF would be violated.

Now, let $v \in X_r$ denote the first node being updated after round r, such that $out_2(v) \notin X_r$ holds, i.e. the new target of node v lies outside X_r. The existence of such a node will be shown below. Let r_v denote the round in which v is updated. We prove that within the round $r, r+1, \ldots, r_v - 1$ all nodes in X_r are only connected to nodes in X_r. We prove this by induction. Initially, at round r, all nodes in X_r are connected to nodes in X_r. Now, consider a round r', with $r \leq r' < r_v - 1$, in which a node w is updated. If w is not contained in X_r, still all nodes in X_r are connected to nodes in X_r as the update of w does not change the forwarding behavior of any node in X_r. Now, if w *is* contained in X_r, then as $r' < r_v$ holds, the new target of w must lie in X_r by the choice of v, i.e. $out_2(w) \in X_r$ holds. Hence, still all nodes in X_r are connected to nodes in X_r by transitivity. Hence, by induction, within the rounds $r, r+1, \ldots, r_v - 1$, all nodes in X_r are connected to nodes in X_r only. This implies the following:

1) Any node $w \in X_r$ leads to a loop in round $r' \in \{r, r+1, \ldots, r_v - 1\}$.
2) Any node $w \in X_r$ is not reachable from the source in round $r' \in \{r, r+1, \ldots, r_v\}$.

To see that the first statement holds, it suffices to note that all nodes in X_r must lead to a loop eventually, as any node has one outgoing neighbor and X_r is finite. The second statement follows from the first: In rounds $r, r+1, \ldots, r_v - 1$ the nodes in X_r are not reachable from the source as this would violate RLF. Furthermore, in round r_v only the update of node v is executed. By updating the rule of v, none of the nodes in X_r can become reachable from the source as node v itself has not been reachable after round $r_v - 1$.

Considering the existence of v, we note the following. The above proof has shown that all nodes in X_r lead to loops in rounds $r, r+1, \ldots, r_v - 1$ and are hence not reachable from the source. If a node v leading outside of X_r was not to exist, then the update schedule cannot be feasible as then loops were to exist even after all nodes are updated.

Given the above, we now adapt the update schedule $\hat{U}$ by iteratively moving the update of node v to a previous round. Concretely, let $\tilde{U}(i)$ denote the update schedule obtained from $\hat{U}$ in which the update of node v is executed in round $r_v - i$ while the updates of rounds $r_v - i, \ldots, r_v - 1$ (with respect to $\hat{U}$) are postponed by a single round. We will prove by induction that the update schedule $\tilde{U}(i)$ does not violate WPE or RLF for $0 \leq i \leq r_v - r$. Initially, for $i = 0$, $\tilde{U}(0) = \hat{U}$ is valid with respect to RLF + WPE. Assuming that $\tilde{U}(i)$ is valid for $0 \leq i < r_v - r$, we now show that $\tilde{U}(i+1)$ satisfies RLF + WPE as well. $\tilde{U}(i+1)$ is obtained from $\tilde{U}(i)$ by swapping the update of node v in round $r_v - i$ with the update executed in round $r_v - i - 1$. Clearly, node v has not been reachable from the source in rounds $r, \ldots, r_v - i$, as $\tilde{U}(i)$ agrees with the update schedule $\hat{U}$ on the first $r_v - i - 1$ rounds. As node v has therefore not been reachable in round $r_v - i - 1$ (with respect to $\tilde{U}(i)$), swapping the update of v with the update of round $r_v - i - 1$ can never violate WPE or RLF as both properties only need to be safeguarded for nodes reachable from the source.

Hence, by induction $\tilde{U}(r_v - r)$, i.e. the update schedule in which the update of node v has been moved from r_v to r while postponing the updates of rounds $r, \ldots, r_v - 1$ by one (all with respect to $\hat{U}$), is still a valid update schedule under $\text{RLF} + \text{WPE}$. Denoting by $\tilde{U}$ the update schedule $\tilde{U}(r_v - r)$, we now prove that $l(\tilde{U}) > l(\hat{U})$ holds, i.e. the first loop according to $\tilde{U}$ is created in a later round than according to $\hat{U}$. We only need to prove that $\tilde{U}$ does not create a loop in round r. Assume for the sake of contradiction that this is not the case, i.e. the update of v in round r creates a loop. As updating v creates a loop, we know that v can be reached from $\mathit{out}_2(v)$ before executing the update of round r. By transitivity, as $v \in X_r$ held by choice of v before the update of round r (note that X_r always references the state before round r) and as $\mathit{out}_2(v)$ can reach v, $\mathit{out}_2(v)$ can also reach x, i.e. $\mathit{out}_2(v) \in X_r$ must hold. This however contradicts the choice of v as we required $\mathit{out}_2(v)$ to be *not* contained in X_r.

Hence, as $\tilde{U}$ is a valid schedule under $\text{RLF} + \text{WPE}$ and does not contain a loop in the first $r + 1$ rounds, we have $l(\tilde{U}) > l(\hat{U})$. This contradicts the choice of $\hat{U}$ being the update schedule that delays the creation of the first loop the longest. Hence, $\max_{U \in \mathcal{U}} l(U) = \infty$ holds for any feasible network update instance, i.e. if there exists a valid update schedule under $\text{RLF} + \text{WPE}$, then there also always exists a valid update schedule satisfying $\text{SLF} + \text{WPE}$. $\square$

The proof of the above theorem is constructive and we obtain the following interesting corollary.

Corollary 1. *Any valid schedule under* $\text{RLF} + \text{WPE}$ *can be turned into a valid schedule under* $\text{SLF} + \text{WPE}$.

Proof. Orienting ourselves at the proof of Theorem 4, we outline an algorithm to obtain a schedule under $\text{SLF} + \text{WPE}$ given a schedule under $\text{RLF} + \text{WPE}$.

1) If multiple updates take place in a single round, spread these updates onto distinct, successive rounds, such that all updates are performed sequentially.
2) Iteratively delay the creation of loops as described in the proof of theorem 4.
3) Merge the updates of adjacent rounds (e.g. greedily) as long as neither WPE nor SLF are violated (optional).

The correctness of the algorithm is immediate: the schedule obtained in Step 1) still satisfies $\text{RLF} + \text{WPE}$, the schedule obtained in Step 2) satisfies $\text{SLF} + \text{WPE}$, and Step 3) cannot violate WPE or SLF. $\square$

IV. NP-Hardness

Our observation that LF and WPE can conflict may not necessarily be a problem in practice: if these instances can be identified quickly, one could resort to alternative, possibly more resource-intensive update mechanisms [6], [30]. Unfortunately, however, as we will prove in the following, the underlying decision problem is NP-hard. In particular, we will prove a polynomial-time reduction from 3-SAT: we construct a network update instance according to a 3-SAT formula which is updatable if and only if the 3-SAT formula is satisfiable. We will refer to the 3-SAT formula as $\mathcal{C}$ and to the respective network update instance as $G(\mathcal{C})$.

Notation: In our reduction we assume that each clause in 3-SAT has exactly 3 literals. We denote the number of variables as k and the number of clauses as m. The variables are denoted as $x_1, x_2, \ldots, x_k$ and the clauses as $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m$. We denote the total number of clauses with variable x_i as m_i, number of clauses with literal x_i as p_i and number of clauses with literal $\neg x_i$ as q_i. We also denote the clauses with literal x_i as $P^i_1, P^i_2, \ldots, P^i_{p_i}$ and the clauses with literal $\neg x_i$ as $Q^i_1, Q^i_2, \ldots, Q^i_{q_i}$.

General Structure: In the constructed instance there will be a destination d, waypoint wp and auxiliary nodes u_1, u_2, u_3 and δ. For each variable and each clause in 3-SAT we will create a gadget. Additionally for each clause we will add three nodes d^i_1, d^i_2, d^i_3 and for each variable we will add m nodes $r^1_j, \ldots, r^m_j$. The source of the path will be the first node in the first clause gadget. The order of gadgets and nodes is presented in Figure 4.

In a gadget for variable x_i there will be a set of nodes connected with clauses containing x_i. Updating one of those edges connecting a clause with the gadget will allow to untangle the corresponding clause (we will define untangling more formally later; generally speaking in order to untangle a clause we need to update one of its edges which corresponds to satisfying it in 3-SAT formula). The variable gadgets will be constructed such that until all clauses are untangled only the edges corresponding to one literal (x_i or $\neg x_i$) can be updated. Therefore the constructed instance will be solvable only if we can untangle all clauses using one literal for each variable, which corresponds to satisfying 3-SAT formula.

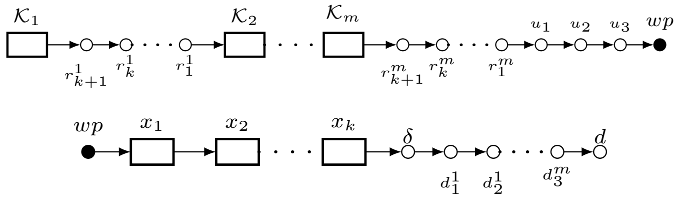


Fig. 4. Order of gadgets and nodes. The upper part shows the order from the source to wp. The lower part shows the order from wp to the destination.

Variable Gadgets: For each variable x_j we construct a gadget (cf. Figure 5 for a visualization). The nodes $v^j_1, v^j_2, v^j_3, v^j_4$ are connected via edges (v^j_1, v^j_3) and (v^j_2, v^j_4) and there is an edge from v^j_4 to the first node of next variable gadget, v^{j+1}_1 (and in case of the last variable gadget there is an edge from v^k_4 to δ). Note that the nodes v^j_4 and v^{j+1}_1 would actually be contracted into a single node in our model, but the proof is simpler when considering these nodes separately. In the gadget there are also nodes $y^j_1, y^j_2, \ldots, y^j_{p_i}$ between v^j_1

and v_2^j and nodes $z_1^j, z_2^j, \dots, z_{q_i}^j$ between v_3^j and v_4^j. The clauses $P_1^j, P_2^j, \dots, P_{p_i}^j$ will be connected to $y_1^j, y_2^j, \dots, y_{p_i}^j$, and updating an edge y_i^j will allow clause P_i^j to become untangled. Similarly clauses $Q_1^j, Q_2^j, \dots, Q_{q_i}^j$ will be connected to $z_1^j, z_2^j, \dots, z_{q_i}^j$. In turn nodes $y_1^j, y_2^j, \dots, y_{p_i}^j$ can be updated only if v_1^j is updated, and $z_1^j, z_2^j, \dots, z_{q_i}^j$ if v_2^j is updated and v_1^j is not updated (or all clauses are already untangled). This will allow us to draw conclusions about the value of x_j based on whether before all clauses become untangled v_1^j is updated or not. A visualization of the gadget structure for variables is presented in .

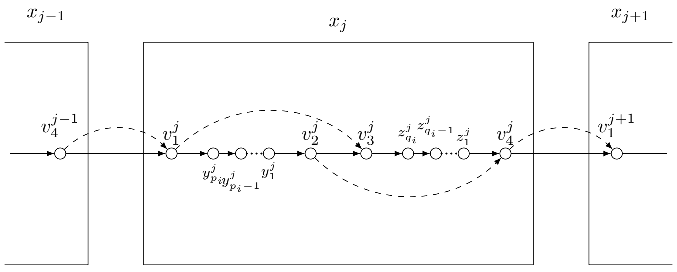


Fig. 5. Construction of a variable gadget for x_j.

Clause Gadgets: For each clause $\mathcal{K}_i$ we construct a gadget consisting of nodes $c_1^i, c_2^i, \dots, c_6^i$. Additionally, we add three nodes (outside of the gadget, close to d, see also Figure 4) d_1^i, d_2^i, d_3^i per clause $\mathcal{K}_i$. For each $j \in \{1,2,3\}$ we add edges (c_j^i, d_j^i) and (d_j^i, c_{j+3}^i). The purpose of nodes d_1^i, d_2^i and d_3^i is to delay the update of nodes c_1^i, c_2^i and c_3^i until all the clauses are untangled. The construction of a clause gadget is shown in Figure 6.

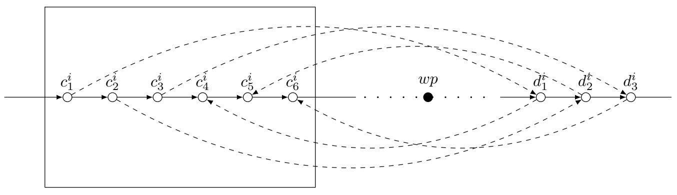


Fig. 6. Construction of a clause gadget.

Connecting the Gadgets: Let us consider variable x_i. Let $\mathcal{K}_j = P_a^i$ be any clause containing literal x_i. Then we connect one of the nodes c_4^j, c_5^j, c_6^j to node y_a^i, and this node to c_1^{j+1}, c_2^{j+1} or c_3^{j+1} (if $\mathcal{K}_j$ is the last clause than to u_1, u_2 or u_3 instead), respectively. Note, that the order of nodes $y_1^i, \dots, y_{p_i}^i$ is reversed compared to the order of $P_1^i, \dots, P_{p_i}^i$ (so the first clause is connected to the last node). We proceed similarly with clauses $Q_1^i, \dots, Q_{q_i}^i$ and nodes $z_1^i, \dots, z_{q_i}^i$. Figure 7 depicts this construction.

Connecting the Whole Graph: In addition to the gadgets we need to connect the path to the destination d, the waypoint node wp and the three nodes u_1, u_2, u_3 which are placed in the old policy just before the waypoint. We add edges from u_3 to wp, from wp to v_1^1, from u_1 to c_2^1 and from u_2 to c_3^1. After every clause gadget we create $k+1$ nodes $r_1^i, r_2^i, \dots, r_{k+1}^i$ in reverse order, i.e. r_{k+1}^i is the first node after the gadget and r_1^i is the last. For each variable x_i we create a path starting in v_3^i,

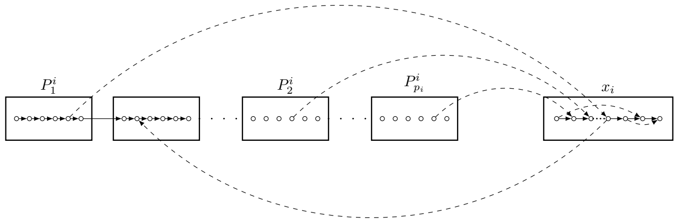


Fig. 7. Edges to connect clauses.

then going through nodes $r_i^m, r_i^{m-1}, \dots, r_i^1$ and ending in v_2^i. We also create a similar path starting in δ, then going through nodes $r_{k+1}^m, r_{k+1}^{m-1}, \dots, r_{k+1}^1$ and ending in v_d^i. All these edges are shown in Figure 8.

Proof of Correctness: In this section we prove, that correctness of the reduction. We say that a clause (or clause gadget) is untangled if at least one of the nodes c_4^i, c_5^i or c_6^i is updated. We say that a clause is tangled if this is not the case.

Theorem 5. *If $\mathcal{C}$ is satisfiable then there is a schedule for $G(\mathcal{C})$ which satisfies* SLF *and* WPE.

Proof. Let $\sigma : \{x_1, \dots, x_k\} \to \{\bot, \top\}$ be an assignment that satisfies $\mathcal{C}$. Based on σ we show how to update all nodes in $G(\mathcal{C})$ without violating SLF or WPE. The nodes will be updated according to the following round schedule:

1) For each variable x_i we update v_2^i. Additionally, if $\sigma(x_i) = \top$ holds, we update v_1^i (which makes the update of v_2^i irrelevant as it bypasses v_2^i).
2) For each variable x_i we update either nodes $y_1^i, \dots, y_{p_i}^i$, if $\sigma(x_i) = \top$ holds, or nodes $z_1^i, \dots, z_{q_i}^i$ otherwise.
3) Since for each clause $\mathcal{K}_j$ there is at least one literal that satisfies it, we update one of the nodes c_4^j, c_5^j, c_6^j connected to that literal. The path after these updates is shown on Figure 9.
4) We update nodes r_j^i for all i, j. This can be done, since every clause has at least one outgoing edge and every r_j^i edge has a clause in between.
5) We update nodes v_3^i, for all i, and node δ, which connects the path updated in round 4 with the reachable parts behind the waypoint.
6) We update those nodes v_1^i that were not updated earlier, as the path starting at v_3^i is now loop-free.
7) We update those nodes y_j^i and z_j^i that were not updated earlier.
8) We update those nodes c_4^j, c_5^j and c_6^j that were not updated earlier.
9) We update nodes d_1^j, d_2^j, d_3^j, for all j.
10) We update nodes c_1^j, c_2^j, c_3^j, for all j.
11) We update nodes u_1, u_2, u_3 and wp.

None of these updates will violate WPE or SLF. □

Theorem 6. *If there is a schedule for $G(\mathcal{C})$ which satisfies* RLF *and* WPE *then $\mathcal{C}$ is satisfiable.*

We will start by proving the following lemma:

Lemma 1. *In any correct order of updating edges, as long as some clause gadgets remain tangled, the following conditions*

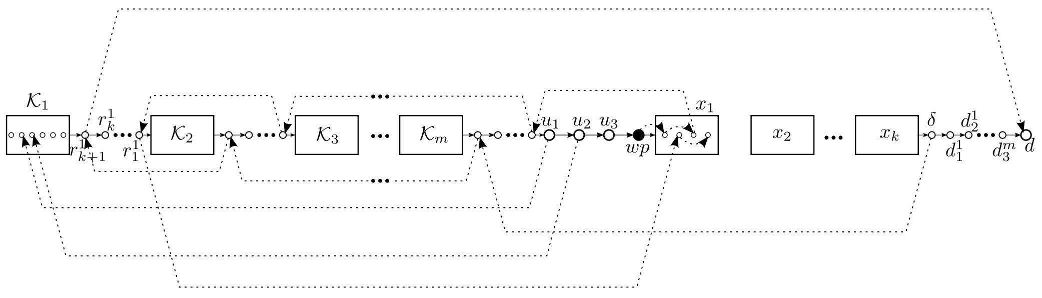


Fig. 8. Connecting all paths.

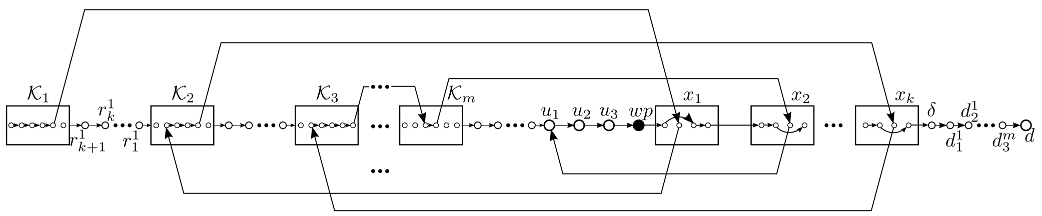


Fig. 9. The path after three rounds of updating according to the schedule in proof of Theorem 5.

hold:

1) *A node y_j^i can be updated only if node v_1^i is updated. A node z_j^i can be updated only if node v_2^i is updated. Nodes z_j^i and v_1^i cannot be both updated.*
2) *For any $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, k+1\}$, node r_j^i can be updated only if the i-th clause gadget is untangled.*
3) *A node c_j^i, for $j \in \{4, 5, 6\}$ can be updated only if its successor is already updated or if there is $h \in \{4, 5, 6\}$ such that $h < j$ and c_h^i is already updated.*
4) *A node v_3^i, for any i, can be updated if r_i^j is updated for all $j \in \{0, 1, \dots, m\}$ or if v_2^i is updated, but v_1^i is not. The same applies to node δ.*
5) *Nodes d_1^i and d_2^i and d_3^i, for any i, cannot be updated.*
6) *Node c_3^i cannot be updated. Node c_2^i can be updated only if c_6^{i-1} and its successor are updated, c_5^{i-1} or its successor are not updated and c_4^{i-1} or its successor are not updated. c_1^i can be updated only if c_4^{i-1} or its successor are not updated and either c_6^{i-1} and its successor or c_5^{i-1} and its successor are updated.*

Before proving the lemma, let us make some observations about what these conditions mean in terms of the path traversed by packets. Conditions 1 and 4 guarantee, that if a packet is in v_1^i or v_2^i, for some i, then it will be forwarded to node δ without going through wp. That is because it uses edges from v_1^j and v_2^j to bypass any backward edges. Then Condition 5 guarantees that it travels from δ to d without passing through the waypoint.

Conditions 2, 3 and 6 guarantee that a packet will traverse from the source through all the clauses until it reaches the waypoint. That holds because for each clause, if it is untangled, the packet will be forwarded from some c_j^i to y_l^a, and then, as y_l^a must have been updated before c_j^i, it returns to c_{j-3}^{i+1}. On the other hand, if the clause is tangled, the packet will go through $r_{k+1}^i, r_k^i, \dots, r_0^i$ (none of them is updated, since the clause is tangled) to c_1^{i+1}.

Conditions 2 and 4 guarantee that as long as not all clauses are untangled, δ cannot be updated and v_3^i can be updated only if the path from source to destination does not go through that node.

Let us also notice that if Conditions 5 and 6 hold, then a packet can enter a clause gadget only through nodes c_1^i, c_2^i and c_3^i, and it is afterwards forwarded to node c_4^i. Hence, it is sufficient to show that a packet enters a clause gadget twice to prove that loop-freedom does not hold.

Proof. Let us take any order of updating edges, and consider the first update that violates one of the conditions. If we update any node other than v_1^i, one of the conditions is violated. So firstly let's assume that only one condition is violated and consider the cases for which condition it is.

1) Let us assume that y_j^i is updated, but v_1^i is not. Then the packet goes through all clauses, and then through all previous variable gadgets. Upon entering the gadget for x_i it goes through an edge from y_j^i to $\mathcal{K}_i$ and therefore violates loop-freedom. The case when z_j^i is updated is similar.
2) Let us assume that r_j^i is updated, but the i-th clause is tangled. Then the packet goes through all clause gadgets up to $\mathcal{K}_i$, then it is forwarded to r_j^i. Then there are two possibilities. Firstly it may be forwarded back to r_l^1, for $l \leq j$, and from there to the gadget for x_l. But because Conditions 1, 4 and 5 are satisfied, it would go to the end, without passing through the waypoint. The other case is that it is forwarded back to r_l^a, for some $a < i$ and $l \leq j$, and then re-enters some clause gadget, which would violate loop-freedom.
3) Let us assume that c_j^i is updated but its successor y_a^l and c_h^i, for all $h \in \{4, 5, 6\}$ such that $h < j$, are not. Then the packet traverses through c_j^i without passing through the waypoint, and then it goes to y_a^l. Then it may either be forwarded to v_2^l, which means that it would be

forwarded to d without passing through the waypoint, because Conditions 1, 4 and 5 are satisfied, or it travels from some node y^l_g to gadget $\mathcal{K}_f$. But then $f \leq i$, because of the order of nodes $y^l_{p_l}, \dots, y^l_1$, it would go to a gadget that was already visited, and therefore violate loop-freedom. The case when the successor of c^i_j is z^l_a, for some l, a, is similar.

4) Let us assume that v^i_3 is updated, but there is some r^j_i which is not updated and either v^i_1 is updated or v^i_2 is not. Then the packet, after going through the waypoint, reaches the gadget for x_i. Then, because v^i_1 is updated or v^i_2 is not, it is forwarded to v^i_3. From there it traverses through some backward edges, before it enters some clause gadget (it cannot take backward edges until it goes to r^1_a, and then go forward to some variable gadget, because Condition 2 holds, and not all clauses are untangled). Since all clause gadgets were already visited, it violates loop-freedom.
5) If d^i_j is updated, then the packet traverses through all clause gadgets and variable gadgets until it reaches d^i_j. From there it goes back to c^i_{j+3}. Then there are two possibilities: if it will be forwarded to the next clause gadget, it will violate loop-freedom, because all clauses were already visited. Otherwise, if c^i_{j+3} is updated, the packet is forwarded to some node y^l_a (or z^l_a). From there, it can either be forwarded to some clause gadget, or to x_{l+1}. In both cases, it violates loop-freedom.
6) The condition guarantees that if the packet traverses through c^i_j, for $j \in \{1, 2, 3\}$, then this node cannot be updated. Otherwise, the packet after going to $\mathcal{K}_i$ (without going through the waypoint), would go to d^i_j, and from there to the destination.

Finally let us consider what happens when we update v^i_1 and it violates Conditions 1 and 4. Then the case is similar to violating only Condition 4, that is, the packet traverses through all the clauses to the waypoint, and from there to v^i_1 and next to v^i_3. From there it goes through some backward edges and re-enters some clause, which violates loop-freedom. □

Now we are ready to prove Theorem 6.

Proof of Theorem 6:. Let us assume that there is a schedule for $G(\mathcal{C})$. Then let us look at the update which untangles the last clause (i.e., before this update there was an tangled clause, and after this update all clauses are untangled). Then Condition 1 guarantees, that for each variable there is no node corresponding to positive literal (node y^i_a) and a node corresponding to negative literal (node z^i_l) that are both updated. This happens as updating node y^i_a requires that v^i_1 is updated, whereas updating node z^i_l requires that v^i_1 is not updated. Therefore we can set the corresponding assignment of variables in $\mathcal{C}$ to $\top$, if at least one of the nodes y^i_a is updated, or to $\bot$ otherwise. Then, as all clauses are untangled, and untangling a clause requires that at least one literal has value $\top$, this assignment satisfies all clauses of $\mathcal{C}$. □

Theorem 7. *The following statements are equivalent:*

1) *The formula $\mathcal{C}$ is satisfiable.*
2) *There is a schedule for $G(\mathcal{C})$ satisfying* RLF + WPE.
3) *There is a schedule for $G(\mathcal{C})$ satisfying* SLF + WPE.

Proof. We have shown that the existence of a schedule satisfying RLF and WPE implies that $\mathcal{C}$ is satisfiable. We have also shown that if $\mathcal{C}$ is satisfiable then there is a schedule satisfying SLF and WPE. Hence, 1) and 2) are equivalent and the equivalence of 3) follows from Theorem 4. □

V. Extension to Service Chains

We currently witness a trend towards more complex network services, which concatenate multiple network functions or middleboxes into so-called *service chains* [11], [26]: sequences of network functions which are allocated and stitched together in a flexible manner. For example, a service chain could define that traffic originating at source s is first steered through an intrusion detection system for security (1^{st} network function), next through a traffic optimizer (2^{nd} network function), and only then is routed towards the destination d.

Clearly, enforcing multiple waypoints does not render the problem easier. Interestingly, it is even *impossible* to compute an update from a route π_1 to a route π_2, if waypoints occur in different order in the two policies.

Theorem 8. *The order, in which two waypoints wp_1 and wp_2 are traversed cannot be changed from π_1 to π_2 without violating either* WPE *or* LF.

Proof. Assume that in π_1 packets traverse wp_1 first, followed by wp_2 and vice versa for π_2. By definition, before the start of the update, packets are forwarded according to π_1 and hence, visit wp_1 before wp_2. Due to WPE, both waypoints are on the source-destination path in every round and hence, to change the order of both waypoints, we can identify a single round where this order changes. Otherwise there are either loops or bypassed waypoints. We can assume w.l.o.g. that this round includes an update, which leads to a forwarding of packets to wp_2 before they traversed wp_1 and that this update will be executed as the first update in this round. However, this update immediately bypasses wp_1, since a way back to wp_1 could only exist if this path existed before. Thus, the round before included a loop, as wp_1 was visited before wp_2. □

Interestingly, however it is still possible to update a policy with multiple waypoints, if only WPE is required (and the waypoints are in the same order in the old policy π_1 and the new policy π_2). Algorithm MultiWayUp is a generalized version of algorithm WayUp. We denote the number of waypoints as k and the waypoints as $wp_1, wp_2, \dots, wp_k$. As in WayUp we will use an upper bound θ on the maximal packet latency. Similarly to the algorithm WayUp we start by updating all nodes, which are only in the new policy π_2. In the second round we update those nodes, which are behind some waypoint wp_i in π_1, but before wp_i in π_2, and which have a backward rule. Then we need k rounds, one for each waypoint, to update nodes before each waypoint in π_2 (i.e., $\pi_2^{<wp_i}$ for each i) in the order of waypoints in π_1. After each round we need to wait θ time to ensure that no packet is on $\pi_1^{<wp_i}$. Finally in the last round we update all remaining packets, i.e., packets of $\pi_2^{>wp_k}$.

Algorithm 2: MultiWayUp

```
1 Input: old policy $\pi_1$, new policy $\pi_2$, threshold $\theta$
2 update nodes of $\pi_2$ which are not in $\pi_1$
3 update
  $\forall i \in [k]$ nodes of $\pi_1^{>wp_i}$ with backw. rules in $\pi_2^{<wp_i}$
4 for $i$ from 1 to $k$
5     update remaining nodes of $\pi_2^{<wp_i}$
6     wait $\theta$
7 update nodes of $\pi_2^{>wp_k}$
```

Theorem 9. MultiWayUp *takes* k+3 *rounds and guarantees the* WPE *property at any time.*

Proof. The round complexity follows from the algorithm definition. The proof of transient consistency is similar to the one in Theorem 2. Line 3 cannot violate WPE, as only forward edges can be used to skip waypoints. Then, in Line 5, we update all nodes in $\pi_2^{<wp_1}$. As all forward nodes, which are updated in this round, lead to nodes in $\pi_2^{<wp_1}$, all packets will eventually use some backward edge updated in Line 3 to reach wp_1. After time θ, all packets reach wp_1 by traversing along π_2, so we can repeat the same argument for the subsequent waypoints. Finally, Line 7 cannot violate WPE, as all packets are already guaranteed to reach all waypoints. □

VI. Optimal Update Algorithms

Given the hardness of the general network update problem, we now present exact algorithms, based on Mixed-Integer Programs (MIPs), for computing update schedules, whenever this is possible. We generalize the Mixed-Integer Program presented in [25] (considering RLF) for multiple waypoints and present the following extensions: (1) we model the decision problem by forcing only a single update to take place in each round, (2) present an adaptation for realizing SLF, and (3) introduce a *flow extension* that computationally strengthens the formulation. Based on these extensions, we obtain 8 different Mixed-Integer Programming formulations in total. We refer to the formulations by 3 character acronyms of the form $-/D\,|\,S/R\,|\,-/F$: the first character indicates whether the decision problem is considered (D) or not (-), the second character indicates whether the strong (S) or the relaxed (R) loop-freedom property is used, and the last indicates whether the flow extension is used (F) or not (-). Hence, DSF refers to the MIP formulation for the decision variant under SLF with the flow extension and -R- denotes the basic MIP formulation for the relaxed-loop freedom property without the decision and flow extensions (cf. MIP 1).

A. Base Formulation

According to the line representation presented in Section III, the policies π_1 and π_2 are paths on the common set of nodes V, such that π_1 and π_2 connect the source $s \in V$ to the destination $d \in V$. We denote the edges of π_1 and π_2 by E_1 and E_2, respectively. We denote by $E = E_1 \cup E_2$ the set of all possible edges. Furthermore, the set of waypoints is denoted by $WP \subseteq V$.

The decision of whether switch $v \in V$ is updated in round $r \in \mathcal{R} = \{1, \ldots, |V| - 1\}$ is modeled using binary variables $x_v^r \in \{0, 1\}$. Constraint 2 of MIP 1 forces the forwarding rule of each node to be changed in exactly one of the rounds. The general objective of the optimization problem is to minimize the number of rounds. This is realized by minimizing the variable $R \geq 0$ which is lower bounded by all the rounds in which an update is performed (see Constraint 1).

Given the assignment of switch updates to rounds, the Constraints 3 and 4 set the variables $y_e^r \in [0, 1]$ to indicate whether the edge $e \in E$ exists after the (successful) execution of all updates up to and including round r. Note that these variables will attain binary values based as these are computed as a function of the binary variables x_v^r. To check that the properties WPE and LF hold, transient states between consecutive rounds need to be considered as discussed below.

a) Enforcing RLF*:* We first outline how to enforce RLF and will then discuss how to adapt the constraints to enforce SLF. To model the RLF property, we need to guarantee that transient states between rounds are loop-free. Note that the updates for round $r \in \mathcal{R}$ will only be triggered when all updates of nodes in previous rounds were successful. As updates within one round are sent out asynchronously, the updates can be installed in an arbitrary order. To effectively forbid any intermediate cycles it is sufficient to forbid cycles in the union of edges already installed after the execution of round $r - 1$ together with the edges that are enabled in round r. This suffices, as, if there exists a partial update of nodes that forms a transient loop, this loop is also contained in the respective union of the edges.

Specifically, considering RLF only loops which are reachable from the source node $s \in V$ need to be considered. To this end, we define variables $a_v^r \in \{0, 1\}$ to indicate whether a node $v \in V$ may be reachable or *accessible* from the source node s under any order of updates between rounds $r - 1$ and r. The variables are set to 1 if, and only if, there exists a (simple) path from s towards v using edges of either the previous round or the current round (see Constraints 5 - 7). Similarly, and based on this reachability information, the variables $y_{u,v}^{r-1\vee r} \in \{0, 1\}$ are set to 1 if the edge $(u, v) \in E$ may be used in the transient state, namely if the edge existed in round $r - 1$ or r and u could be reached (see Constraints 8 and 9). Lastly, having reconstructed the information which edges effectively *may* carry flow, we employ the well-known Miller-Tucker-Zemlin constraints (see Constraint 10) with corresponding level variables $l_v^r \in [0, |V| - 1]$ to forbid loops: if traffic may be sent along edge $(u, v) \in E$, i.e., if $y_{u,v}^{r-1\vee r} = 1$ holds, then $l_v^r \geq l_u^r + 1$ is enforced. Hence, the level variable of the head v of the edge (u, v) is strictly larger than the level variable of its predecessor u. Clearly, an existing cycle does not allow for a feasible assignment of level variables.

We lastly note, that we need to introduce the following constants, modeling the initial state in round 0, for the MIP 1 to be well-defined: we set $a_s^0 = 1$, as the source node s is always reachable, and enforce that initially only edges of the old policy E_1 may be used, i.e. we set $y_{u,v}^0 = 1$ for $(u, v) \in E_1$ and $y_{u,v}^0 = 0$ for $(u, v) \in E_2$.

Mixed-Integer Program 1: Basic Formulation (-R-)

$$\begin{array}{lll}
\min R & & \text{(Obj)} \\
R \geq r \cdot x_v^r & \forall r \in \mathcal{R}, v \in V & (1) \\
1 = \sum_{r \in \mathcal{R}} x_v^r & \forall v \in V & (2) \\
y_{u,v}^r = 1 - \sum_{r' \leq r} x_u^{r'} & \forall r \in \mathcal{R}, (u,v) \in E_1 & (3) \\
y_{u,v}^r = \sum_{r' \leq r} x_u^{r'} & \forall r \in \mathcal{R}, (u,v) \in E_2 & (4) \\
a_s^r = 1 & \forall r \in \mathcal{R} & (5) \\
a_v^r \geq a_u^r + y_{u,v}^{r-1} - 1 & \forall r \in \mathcal{R}, (u,v) \in E & (6) \\
a_v^r \geq a_u^r + y_{u,v}^r - 1 & \forall r \in \mathcal{R}, (u,v) \in E & (7) \\
y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^{r-1} - 1 & \forall r \in \mathcal{R}, (u,v) \in E & (8) \\
y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^r - 1 & \forall r \in \mathcal{R}, (u,v) \in E & (9) \\
y_{u,v}^{r-1 \vee r} \leq \dfrac{l_v^r - l_u^r - 1}{|V| - 1} + 1 & \forall r \in \mathcal{R}, (u,v) \in E & (10) \\
\overline{a}_s^{r,w} = 1 & \forall r \in \mathcal{R}, w \in \mathit{WP} & (11) \\
\overline{a}_v^{r,w} \geq \overline{a}_u^{r,w} + y_{u,v}^{r-1} - 1 & \begin{array}{l}\forall r \in \mathcal{R}, w \in \mathit{WP}, \\ (u,v) \in E_{\overline{\mathit{WP}}}^{w}\end{array} & (12) \\
\overline{a}_v^{r,w} \geq \overline{a}_u^{r,w} + y_{u,v}^{r} - 1 & \begin{array}{l}\forall r \in \mathcal{R}, w \in \mathit{WP}, \\ (u,v) \in E_{\overline{\mathit{WP}}}^{w}\end{array} & (13) \\
\overline{a}_d^{r,w} = 0 & \forall r \in \mathcal{R}, w \in \mathit{WP} & (14)
\end{array}$$

b) Enforcing WPE*:* For enforcing the WPE property, a reachability construction similar to the one of Constraints 5 - 7 is employed. We define variables $\overline{a}_v^{r,w} \in \{0,1\}$ for each waypoint $w \in \mathit{WP}$, each round $r \in \mathcal{R}$ and each node $v \in V$. Intuitively, $\overline{a}_v^{r,w} = 0$ may only hold, if no path from the source towards the node v exists in the transient state between rounds r and $r-1$, which does not contain waypoint $w \in \mathit{WP}$. To this end, we denote by $E_{\overline{\mathit{WP}}}^{w} \subset E$ all edges *not* incident to the waypoint w and reachability propagation is only enforced along these edges (cf. Constraints 11 - 13). As Constraint 14 ensures that no packet must arrive at the destination d – using a path in $E_{\overline{\mathit{WP}}}^{w}$ – no waypoint $w \in \mathit{WP}$ will be bypassed in any transient state.

B. Model Extensions

Based on MIP 1 for RLF and WPE, we now consider a series of model extensions.

1) Decision Variant: First, note that the above presented formulation only considers the *optimization* problem of finding an update schedule using the minimal number of rounds. However, for checking whether a given problem is feasible or not, it will prove useful to consider the respective decision problem. To this end, we may include the following constraint, which allows only one switch to be updated per round.

$$\sum_{v \in V} x_v^r = 1 \quad \forall r \in \mathcal{R}\,. \tag{15}$$

While simple, this constraint can drastically reduce the search space and acts as a *symmetry reduction.*

2) Enforcing SLF*:* SLF is strictly stronger than RLF as it forbids cycles under any circumstances, i.e. it forbids cycles even if none of the nodes on the cycle are (anymore) reachable from the source node. Hence, to obtain the formulation for SLF, we may simply assume that all nodes are always reachable from the source. Hence, fixing all variables a_v^r to 1, the Constraints 5 - 9 reduce to:

$$\begin{array}{lll}
y_{u,v}^{r-1 \vee r} \geq y_{u,v}^{r-1} & \forall r \in \mathcal{R}, (u,v) \in E & (16) \\
y_{u,v}^{r-1 \vee r} \geq y_{u,v}^{r} & \forall r \in \mathcal{R}, (u,v) \in E & (17)
\end{array}$$

Note that the reachability variables a_v^r are not used anywhere else and hence these do not need to be instatiated for SLF.

3) Flow Extension: A disadvantage of the MIP 1 is the (necessary) use of reachability propagation constraints in the form of binary conjunctions (cf. Constraints 6 - 9): *if* the tail u is reachable *and* the respective edge (u,v) is enabled, *then* the head v is also reachable. These constraints often yield weak linear relaxations [4], which may lead to high runtimes in practice. To strengthen the models, we present an extension built on multi-commodity flows. Concretely, we consider s-d flows for each round $r \in \mathcal{R}$ to enforce the correctness of the *non-transient* states and introduce flow variables $f_e^r \in [0,1]$ for each round $r \in \mathcal{R}$ and each edge $e \in E$. Our extension can be formalized as follows:

$$\begin{array}{lll}
\sum_{e \in \delta^+(s)} f_e^r = 1 & \forall r \in \mathcal{R} & (18) \\
\sum_{e \in \delta^+(v)} f_e^r = \sum_{e \in \delta^-(v)} f_e^r & \forall r \in \mathcal{R}, v \in V \setminus \{s,d\} & (19) \\
f_e^r \leq y_e^r & \forall r \in \mathcal{R}, e \in E & (20) \\
\sum_{e \in \delta^-(w)} f_e^r \geq 1 & \forall r \in \mathcal{R}, w \in \mathit{WP} & (21) \\
a_v^r \geq f_v^{r-1} & \forall r \in \mathcal{R} & (22^*) \\
a_v^r \geq f_v^r & \forall r \in \mathcal{R} & (23^*)
\end{array}$$

The first two constraints induce an s-d flow, such that the flow starting at the source $s \in V$ must reach the destination $d \in V$, where $\delta^+(v) = \{(v,u) | (v,u) \in E\}$ and $\delta^-(v) = \{(u,v) | (u,v) \in E\}$ denote the outgoing and incoming edges of v with respect to the edge set E, respectively. As the flow is upper bounded by the existence of the edges (see Constraint 20), not even fractional cycles may exist after having executed the updates of round $r \in \mathcal{R}$; this is in fact not safe-guarded by the linear relaxations of MIP 1. Note, that the Constraint 20 is valid both for SLF and RLF, since the flow always emerges at s and all nodes on the flow are therefore reachable.

With respect to WPE, the Constraint 21 states that all waypoints must be reached by all of the flow by lower bounding the flow along the the set of incoming edges. Lastly, the Constraints 22* and 23* strenghten the (relaxed) loop detection by bounding the reachability variables a_v^r from below. Note that these constraints are only added, when RLF is considered, as these variables do not exist in the SLF variant.

VII. Simulations

We generated instances (i.e., scenarios) with varying number of nodes and waypoints and are specifically interested in the number of scenarios in which LF and WPE conflict, and hence

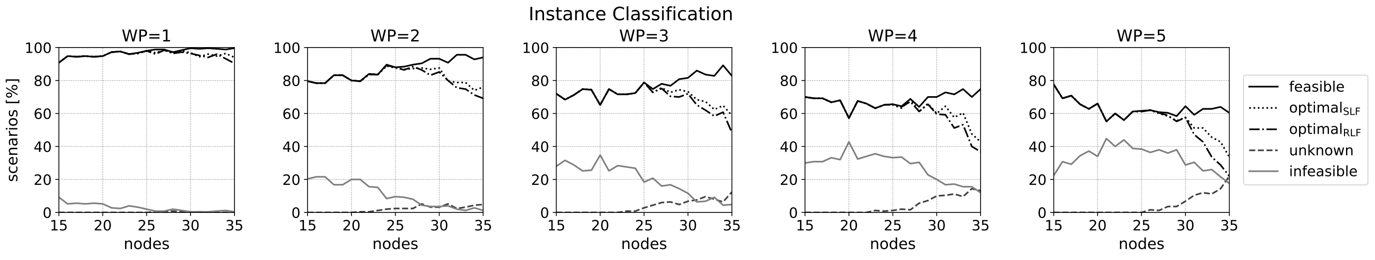


Fig. 10. Classification of the generated scenarios according to whether a feasible solution exists, how many optimal solutions can be computed under strong and relaxed loop-freedom, how many scenarios are infeasible and lastly for which percentage of the scenarios neither feasibility nor infeasibility can be proven with any of the algorithms in 1,000 seconds. As can be seen, the number of waypoints has a distinct impact on the number of feasible scenarios.

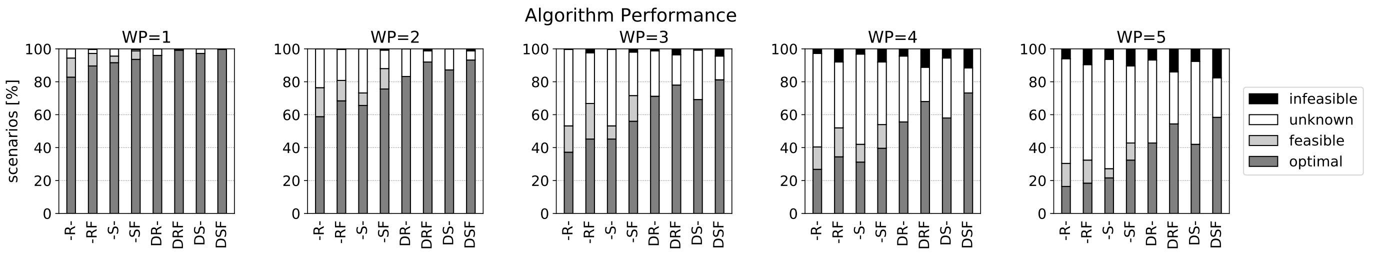


Fig. 11. Qualitative evaluation of the studied algorithms for the maximum number of 35 switches. For each of the eight algorithms and each of the 250 instances (per waypoint) the output is classified as in Figure 10. When considering the *decision variant*, feasibility coincides with optimality.

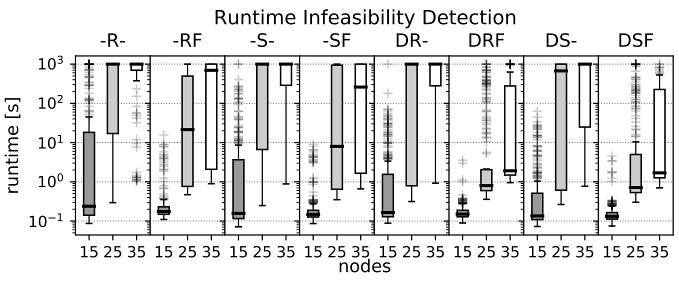
are not updateable without violating either. We generate policy updates randomly and vary the number of nodes as well as the number of waypoints. The generated policies always have a fixed source node $s \in V$ and destination node $d \in V$, and the intermediate nodes' order is shuffled uniformly at random. In cases of multiple adjacent waypoints, we ensure that the order in which the waypoints are traversed does not differ. This is necessary, as by Theorem 8 these scenarios are unsolvable (and can easily be identified). In addition, we guarantee that the policies are (edge-wise) disjoint, as the respective tail nodes do not need to be updated in the first place.

We generate updates of number of nodes $\{15, \ldots, 35\}$, containing one to five waypoints. For each combination of these values, we generate 250 instances at random (26,250 overall). Our Python2.7 implementation uses Gurobi 8.0 to solve the respective eight different MIP formulations and terminate experiments, if neither optimality nor infeasibility was shown, after 1,000 seconds. The complete code for our evaluation, including the instances studied and our results, is freely available at [31]. The experiments were conducted on a server equipped with Intel Xeon E5-4627v3 CPUs running at 2.60GHz.

As to be expected, the feasibility of scenarios is influenced by the number of waypoints within the update (cf. Figure 10). For a single waypoint the percentage of infeasible scenarios is decreasing when increasing the number of nodes. While roughly 10% of the scenarios are infeasible for 15 nodes, for scenarios involving at least 28 nodes only a fraction of at most 3% remains infeasible. These numbers increase significantly when considering more waypoints, leading to close to 20% infeasible instances when considering 5 waypoints and 35 nodes. More waypoints do not only lead to a higher percentage of infeasible scenarios, but also to more time-intensive computations. While scenarios with one waypoint are solved in nearly all cases within the 1,000 seconds, the feasibility of roughly 20% of the scenarios cannot be decided when considering 5 waypoints at 35 nodes, i.e., within the time limit neither the feasibility nor the infeasibility could be proven using any of the Mixed-Integer Programs. Furthermore, the number of scenarios solved to optimality decreases both with the scenario size and the number of waypoints. Comparing the different notions of Loop-Freedom, we note that for RLF a smaller amount of scenarios could be solved to optimality.

The feasibility trends observed in Figure 10 can also be found in Figure 11, which gives an overview of the algorithms' performance for the scenarios containing 35 nodes. Independent of the number of waypoints, it can be observed that the MIP variants, which only allow a single update per round (i.e. the decision variants), find more solutions than the formulations which aim for a minimization of the number of rounds. Note that when considering the decision variant, any feasible solution is an optimal solution. The flow extension formulations also show a slight benefit compared to its counterparts in terms of the ability to find feasible solutions as well as to detect infeasibilities. However, in scenarios with multiple waypoints, some optimization variants fail to detect a significant fraction of up to 20% of feasible scenarios as such, i.e. even though solutions exist, none are found within the time limit.

Figure 12 provides a more detailed view on the runtime distributions leading to the increase of undecided scenarios. While there are only negligible differences for 15 nodes, these differences increase drastically with the number of nodes. In terms of the infeasibility detection, there is a significant improvement when adding the flow extension to the algorithms, e.g., the median runtime for the 25 node scenarios decreases roughly by two orders of magnitude (specifically, when considering the decision variants). When considering the time until the first solution was found, no clear impact can be detected. The algorithm yielding solutions the fastest is the



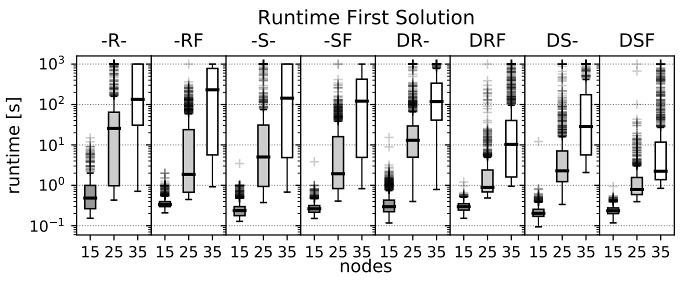


Fig. 12. Runtimes for proving the infeasibility of the underlying scenario (*left*) and for finding the first feasible solution (*right*) aggregated over all number of waypoints and displayed for 15, 25, and 35 nodes as well as for each Mixed-Integer Programming formulation. Note the logarithmic y-axis.

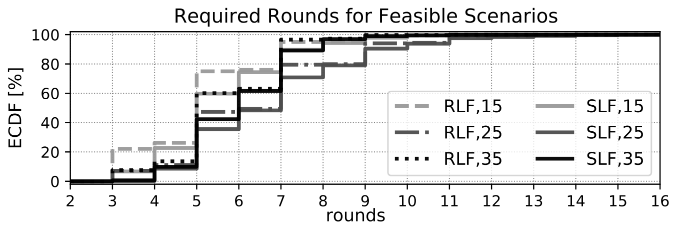


Fig. 13. ECDF of the minimal number of required rounds for updating (*feasible*) scenarios with 15, 25, and 35 nodes under RLF (*dashed/dotted*) and SLF (*solid*). More than 90% of the scenarios can be solved using 10 or less rounds. Note the logarithmic x-axis.

strongest formulation (with respect to its LP relaxations): the decision variant for SLF using the flow extension: even for 35 nodes the majority of solutions is found within less than 10 seconds. Given these results one could possibly use the decision variant to construct a solution quickly (or decide that no solution exists) and – if a solution was found – use the found solution to warm-start the Mixed-Integer Program for minimizing the rounds.

We lastly present in Figure 13 we show the difference between RLF and SLF, in terms of the We consider scenarios with 15, 25, and 35 nodes. We observe that more than 90% of the scenarios are solvable using at most 10 rounds. Furthermore, the usage of RLF decreases the required number of rounds by roughly one on average.

VIII. Related Work

The problem of how to consistently update network routes has received much attention over the last years, also motivated by the advent of software-defined networks. A recent survey of the field [13] classifies network update problems according to the consistency property they provide: *connectivity* (e.g., blackhole- and loop-freedom [2], [12], [14], [15]), *policy* (e.g., per-packet consistency [8], [30] or waypoint enforcement), and *performance* (taking into account the actual availability and limits of network bandwidth resources, e.g., updated must be congestion-free [5], [17]). We in this paper initiated the study of a fundamental policy-aware update problem: updates which also respect waypoints.

In their seminal work, Reitblatt et al. [30] initiated the study of SDN route updates providing strong, per-packet consistency guarantees. Their 2-phase commit protocol also forms the basis of the distributed implementation in [6]. Mahajan and Wattenhofer [27] started investigating weaker transient consistency properties for destination-based routing policies, and introduced the *node-ordering technique*. The model introduced in [27] already led to several follow-up papers, for example [12] which also studies round-based models and [10] which studies extensions to jointly optimizing multiple policies. Our work builds upon [27]. Besides considering a new consistency property, waypoint enforcement, and we also go beyond destination-based routing.

Researchers have also started investigating consistent updates for networks which include (network function virtualized) middleboxes. In their interesting work [16], Ghorbani and Godfrey argue that in the context of network function virtualization, rather stronger consistency properties are required. In terms of more general policy-respecting updates, McClurg et al. [28] present a model for consistent updates using Linear Temporal Logics and show how to synthesize SDN updates automatically. In a recent work, Vissicchio et al. [32] propose an algorithm for preserving generic policies during SDN updates. Here, a policy is defined as a set of paths such that the flow must traverse any of those paths in each intermediate state.

The measurement studies in [20] and [23] provide empirical evidence for the non-negligible time and high variance of switch updates, motivating our work. However, there is also work on network update problems in synchronous models: Time4 [29] makes stronger assumptions on the synchronization in the network and among switches, which however leads to a very different algorithmic problem [33].

Finally, there is also first work on polynomial-time update algorithms for special cases (e.g., [8]), in particular on acyclic graphs [3], however, without focusing on waypoint properties.

Bibliographic note. First versions of this work have been presented at the ACM HotNets 2014 workshop [25] and at the ACM SIGMETRICS 2016 conference [24].

IX. Conclusion

This paper initiated the study of how to update a network in a transiently consistent manner, ensuring not only classic forwarding correctness properties related to connectivity (namely loop-freedom), but also respecting network (security) policies and in particular waypoint enforcement. We believe that our paper opens a rich and interesting area of research. In particular, it would be interesting to further explore polynomial-time algorithms like [3], for special instances of the network update problem.

Acknowledgments We would like to thank Damien Foucard and Apoorv Shukla for many inputs and feedback.

References

[1] A. Gember-Jacobson et al. OpenNF: Enabling innovation in network function control. In *Proc. ACM SIGCOMM*, 2014.

[2] S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. Transiently consistent sdn updates: Being greedy is hard. In *Proc. SIROCCO*, 2016.

[3] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht. Congestion-free rerouting of flows on dags. In *Proc. 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2018.

[4] D. Bertsimas and R. Weismantel. *Optimization over integers*. Dynamic Ideas Belmont, 2005.

[5] S. Brandt, K.-T. Förster, and R. Wattenhofer. On consistent migration of flows in sdns. In *Proc. IEEE INFOCOM 2016*, 2016.

[6] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. IEEE INFOCOM*, 2015.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[8] P. Černỳ, N. Foster, N. Jagnik, and J. McClurg. Optimal consistent network updates in polynomial time. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 114–128, 2016.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, 2011.

[10] S. Dudycz, A. Ludwig, and S. Schmid. Can't touch this: Consistent network updates for multiple policies. In *Proc. 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[11] ETSI. Network functions virtualisation (nfv); use cases. *http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf*, 2014.

[12] K.-T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid. Loop-free route updates for software-defined networks. In *Proc. IEEE/ACM Transactions on Networking (ToN)*, 2018.

[13] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of Consistent Software-Defined Network Updates. *Technical Report, arXiv*, 2018.

[14] K. Förster and R. Wattenhofer. The power of two in consistent network updates: Hard loop freedom, easy flow migration. In *Proc. ICCCN*, 2016.

[15] K.-T. Förster, R. Mahajan, and R. Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *Proc. IFIP Networking*, 2016.

[16] S. Ghorbani and B. Godfrey. Towards correct network virtualization. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2014.

[17] H. Harry et al. zUpdate: Updating Data Center Networks with Zero Loss. In *Proc. ACM SIGCOMM*, 2013.

[18] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *Proc. ACM SIGCOMM*, 2013.

[19] J. Sherry et al. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, 2012.

[20] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang. Dionysus: Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[21] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proc. ACM CoNEXT*, pages 13–24, 2013.

[22] M. Kuzniar, P. Peresini, and D. Kostić. Providing reliable fib update acknowledgments in sdn. In *Proc. ACM CoNEXT*, 2014.

[23] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about sdn flow tables. In *Proc. Passive and Active Measurement (PAM)*, pages 347–359. Springer, 2015.

[24] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *Proc. ACM SIGMETRICS*, 2016.

[25] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[26] T. Lukovszki and S. Schmid. Online admission control and embedding of service chains. In *Proc. SIROCCO*, 2015.

[27] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proc. ACM HotNets*, 2013.

[28] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proc. ACM SIGPLAN PLDI*, 2015.

[29] T. Mizrahi and Y. Moses. Time4: Time for sdn. *IEEE Transactions on Network and Service Management*, 13(3):433–446, 2016.

[30] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

[31] M. Rost. Source code for the evaluation presented in this work. https://github.com/MatthiasRost/NetworkUpdateScheduler.

[32] S. Vissicchio and L. Cittadini. Safe, efficient, and robust SDN updates by combining rule replacements and additions. *IEEE/ACM Trans. Netw.*, 25(5):3102–3115, 2017.

[33] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, and J. Wu. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *Proc. IEEE ICDCS*, 2018.

Arne Ludwig received his Diploma and PhD in Computer Science from the Technical University of Berlin, Germany in 2011 and 2016. He is currently working at the SAP Innovation Center Potsdam. His research interests include (Online-)Algorithms, SDN, Network Virtualization and Network Economics.

Szymon Dudycz Szymon Dudycz is a PhD candidate at University of Wroclaw. He received his MSc in computer science at University of Wroclaw at 2016. His main research interests are various matching variants and their application in approximation algorithms.

Matthias Rost is a PhD candidate at TU Berlin, Germany. He received his MSc in computer science at TU Berlin, Germany, and was awarded the KuVS prize 2014 for his master thesis on computing virtual aggregation and multicast trees by the German Informatics Society. His main research focus is the theoretical design of provably good algorithms for service orchestration in networks and their application in practice. For his work on the complexity of virtual network embeddings, he received the IFIP NETWORKING 2018 best paper award.

Stefan Schmid is Professor for Computer Science at the University of Vienna, Austria. He received his MSc (2004) and PhD (2008) from ETH Zurich, Switzerland. Subsequently, Stefan Schmid worked as postdoc at TU Munich and the University of Paderborn (2009). From 2009 to 2015, he was a senior research scientist at the Telekom Innovations Laboratories (T-Labs) in Berlin, Germany, and from 2015 to 2018, an Associate Professor at Aalborg University, Denmark. His research interests revolve around the fundamental algorithmic problems of networked and distributed systems.