Transiently Secure Network Updates

Arne Ludwig
TU Berlin, Germany
arne@inet.tu-berlin.de

Szymon Dudycz
University of Wroclaw, Poland
szymon.dudycz@gmail.com

Matthias Rost
TU Berlin, Germany
mrost@inet.tu-berlin.de

Stefan Schmid
Aalborg University, Denmark
schmiste@cs.aau.dk

ABSTRACT

Computer networks have become a critical infrastructure. Especially in shared environments such as datacenters it is important that a correct, consistent and secure network operation is guaranteed at any time, even during routing policy updates. In particular, at no point in time should it be possible for packets to bypass security critical waypoints (such as a firewall or IDS) or to be forwarded along loops.

This paper studies the problem of how to change routing policies in a transiently consistent manner. Transiently consistent network updates have been proposed as a fast and resource efficient alternative to per-packet consistent updates. Our main result is a negative one: we show that there are settings where the two basic properties waypoint enforcement and loop-freedom cannot be satisfied simultaneously. 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 compute optimal update schedules. We report on extensive simulation results and initiate the discussion of scenarios where multiple waypoints need to be ensured (also known as service chains).

1. 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* [5], fast and in a globally consistent manner.

However, today, we do not have a good understanding yet 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 [11, 13, 19, 22, 31]. 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 [6, 13, 18]. The non-atomicity of seemingly atomic data plane updates means that there are periods when the network configuration is incorrect despite looking correct from the control plane perspective [17].

In a first line of works, initiated by Reitblatt et al. [31], 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 [22], 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 the most basic transient guarantees, such as *Loop-Freedom* (LF): packets will never be forwarded along a loop. Transiently loop-free updates are an attractive alternative to stronger consistency models such as PPC, as updates take effect earlier, are more resource efficient (no need for extra rules on the switch), and do not require tagging (often problematic given the limited packet header space). Ludwig et al. [19] have observed that besides the canonical *strong* loop-freedom, there exists a *relaxed* notion of loop-freedom which facilitates even faster network updates.

Security critical environments require additional consistency guarantees, beyond loop-freedom. A particularly important one is *Waypoint Enforcement* (WPE): it needs to be ensured that packets traverse specific network functions or middleboxes. 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 [14]. In order to enforce such a policy, traffic from s to d needs to traverse a middlebox instance inspecting and classifying the flows. The number of middleboxes in enterprise networks can be of the same order of magnitude as the number of routers [12]. Middleboxes can also be virtual-

SIGMETRICS '16, June 14–18, 2016, Antibes Juan-Les-Pins, France.

ISBN 978-1-4503-4266-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2896377.2901476

ized [1]: such middleboxes can be deployed fast and flexibly on the virtualized network nodes, e.g., running in a virtual machine on a commodity x86 server.

In a nutshell, a transiently consistent waypoint enforcing upate should guarantee that before, during and after the transition from π_1 to π_2, packets traverse a certain middlebox (the waypoint).

1.1 Contributions

This paper studies the problem of how to consistently update a network such that waypoint enforcement and loop-freedom are ensured. In particular, we make the following contributions:

1. **Waypoint enforcement matters:** We show that waypoints may easily be bypassed 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 show that the transient Waypoint Enforcement WPE property may conflict with Loop-Freedom (LF), in the sense that both properties may not be implementable simultaneously. We also prove that relaxing the notion of loop-freedom, as suggested by Ludwig et al. [19] for performance reasons, does not help to render impossible instances feasible: a given problem instance which cannot be solved under strong loop-freedom, cannot be solved with relaxed loop-freedom 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 the relaxed loop-freedom.

4. **Multiple waypoints:** We initiate the discussion of how to consistently update routes with more than one waypoint, a.k.a. *service chains*. In particular, we show that flexibilities in the order in which service chain functions are traversed do not help regarding feasibility.

5. **Algorithms:** We present optimized, exact mixed integer algorithms for the different network update variants. While solving mixed integer programs can be resource- and time-consuming in general, we show that in some scenarios, solutions can be computed efficiently, and present a succinct formulation accordingly.

6. **Simulations:** We report on an extensive simulation study. In particular, we find that the scenario size 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.

1.2 Paper Organization

The remainder of this paper is organized as follows. Section 2 introduces our formal model. Section 3 presents first intuitions and insights. The formal NP-hardness proof is given in Section 4. Section 5 initiates the discussion of multiple waypoints. Section 6 presents different optimized and exact algorithms (mixed integer programs). Simulation results are discussed in Section 7. After reviewing related work in Section 8, we conclude our contribution in Section 9.

2. FORMAL MODEL

We consider a network which is managed by a controller, sending out updates to the different switches across an asynchronous network. As the updates are asynchronous, we require the controller to send out simultaneous updates only to a "safe" subset of nodes. Only after these updates have been confirmed (using acknowledgments), the next subset is updated.

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 are simple directed paths. Initially, packets are forwarded (using the *old rules*, henceforth also called *old edges*) along π_1, 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 or none of the two paths are trivially updatable, 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 $\pi_1 = (s = v_0, v_1, \ldots, v_{\ell-1} = d)$ and $\pi_2 = (s = v_0, \pi(v_1), \ldots, \pi(v_{\ell-2}), v_{\ell-1} = d)$, for some permutation $\pi : V \setminus \{s,d\} \to V \setminus \{s,d\}$ and some number ℓ. In fact, we can represent policies in an even more compact way: we are actually only concerned about the nodes $U \subseteq V$ which need to be updated. Let, for each node $v \in V$, $out_1(v)$ (resp. $in_1(v)$) denote the outgoing (resp. incoming) edge according to policy π_1, and $out_2(v)$ (resp. $in_2(v)$) denote the outgoing (resp. incoming) edge according to policy π_2. Moreover, let us extend these definitions for entire node sets S, i.e., $out_i(S) = \bigcup_{v \in S} out_i(v)$, for $i \in \{1,2\}$, and analogously, for in_i. We define s to be the first node (say, on π_1) with $out_1(v) \neq out_2(v)$, and d to be the last node with $in_1(v) \neq in_2(v)$. We are interested in the set of to-be-updated nodes $U = \{v \in V : out_1(v) \neq out_2(v)\}$, and define $n = |U|$. Given this reduction, in the following, we will assume that V only consists of interesting nodes ($U = V$).

Our main objective is to compute update schedules which ensure loop-freedom and waypoint enforcement. We will discuss the two properties in turn.

2.1 Loop-Freedom

We distinguish between *Strong Loop-Freedom* and *Relaxed Loop-Freedom* [19].

2.1.1 Strong Loop-Freedom (SLF)

Our goal is to find an *update schedule* $U_1, U_2, \ldots, U_k$, i.e., a sequence of subsets $U_t \subseteq U$ where the subsets form a partition of U (i.e., $U = U_1 \cup U_2 \cup \ldots \cup U_k$), with the property that for any round t, given that the updates $U_{t'}$ for $t' < t$ have been made, all updates U_t can be performed "asynchronously", that is, in an arbitrary order without violating

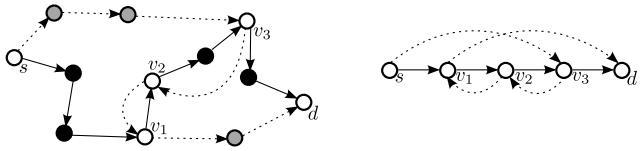

Figure 1: Overview of model and reduction (adopted from [19]). The solid lines show the old policy π_1 and the dashed lines show the new policy π_2. The update problem on the *left* boils down to the update problem for the line representation depicted on the *right* (the old route goes from left to right). Nodes shown in white are the only ones which are part on both paths, and hence relevant for the problem.

loop-freedom. Thus, consistent paths will be maintained for any subset of updated nodes, independently of how long individual updates may take.

More formally, 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. Since updates during round t occur asynchronously, an arbitrary subset of nodes $X \subseteq U_t$ may already have been updated while the nodes $\overline{X} = U_t \setminus X$ still use the old rules, resulting in a temporary forwarding graph $G_t(U, X, E_t)$ over nodes U, where $E_t = out_1(U_{>t} \cup \overline{X}) \cup out_2(U_{<t} \cup X)$. We require that the update schedule $U_1, U_2, \ldots, U_k$ fulfills the property that for all t and for any $X \subseteq U_t$, $G_t(U, X, E_t)$ is loop-free.

Figure 1 illustrates our model [19]: We are given two policies (the old rules of π_1 are *solid*, the new ones of π_2 are *dashed*), see Figure 1 (*left*). We focus on the updateable nodes which are shared by the two policies. Thus, in our example, the update problem can be reduced to the 5-node chain graph in Figure 1 (*right*).

In the following we will call an edge (u, v) of the new policy π_2 *forward*, if v is closer (with respect to π_1) to the destination, resp. *backward*, if u is closer to the destination. It is also convenient to name nodes after their outgoing dashed edges (e.g., *forward* or *backward*); similarly, it is sometimes convenient to say that we *update an edge* when we update the corresponding node.

2.1.2 Relaxed Loop-Freedom (RLF)

Relaxed Loop-Freedom is motivated by the practical observation that transient loops are not very harmful if they do not occur between the source s and the destination d. If relaxed loop-freedom is preserved, only a constant number of packets can loop: we will never push new packets into a loop "at line rate". In other words, even if nodes acknowledge new updates late (or never), new packets will not enter loops. Concretely, and similar to the definition of SLF, we require the update schedule to fulfill the property that for all rounds t and for any subset X, the temporary forwarding graph $G_t(U, X, E'_t)$ is loop-free. The difference is that we only care about the subset E'_t of E_t consisting of edges *reachable from the source* s.

Throughout this paper we refer to Loop-Freedom (LF) whenever the results hold for both SLF and RLF.

2.2 Waypoint Enforcement

In addition to loop-freedom, we want to ensure waypoint-enforcement (WPE). That is, we assume that both the old route π_1 as well as the new route π_2 traverse a special node in the network, henceforth called the *waypoint*. For instance, the waypoint could describe a firewall or intrusion detection system which each packet should traverse: both before, during, as well as after the update.

2.3 Fast Network Updates

We are interested in consistent network updates which are fast: we want to minimize the number of update rounds k in the schedule, where in each round, the controller sends another batch of updates to a subset of nodes, and waits for their completion before starting the next round. This is a natural metric, given the time it takes to update an individual OpenFlow switch today [6, 13, 18].

3. FIRST OBSERVATIONS

To acquaint ourselves with the problem, in this section, we give some examples and also derive first insights.

3.1 Example

Let us consider the example illustrated in Figure 2. The old policy π_1 connects four nodes (switches), from left to right (depicted as a *straight, solid line*, $s \to v_1 \to v_2 \to d$); 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 temporary forwarding path may emerge 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*.

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.

We, in this paper, are interested in consistent network updates which are *fast*: (parts of) the new paths should be used as soon as possible during the update. Concretely, we want to minimize the *round complexity* of the policy update: the number of communication rounds where in each round, the controller sends another batch of updates to a subset of nodes, and waits for their completion before starting the next round.

3.2 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",

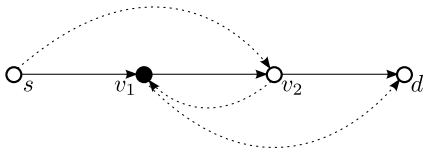

Figure 2: Updating all nodes in one round may violate WPE.

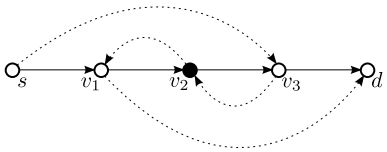

Figure 3: WPE and LF may conflict.

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 will 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 risking the 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. Assume the following update strategy: 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. This problem also exists for any other update strategy. □

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 any packet is actually waypoint enforced.

3.3 Observation 2: It may be impossible!

We next show a negative result: WPE and LF may conflict, i.e., it is sometimes impossible to simultaneously guarantee both properties.

THEOREM 2. *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 risking the 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 again. Hence there is no update schedule which does not violate either WPE or LF. □

3.4 Observation 3: It does not help to relax!

Given the additional flexibility introduced by relaxed loop-freedom, one could hope that relaxed loop-freedom can also help to overcome the impossibility observed above. However, as we prove in the following, this is never the case.

THEOREM 3. *If there does not exist a strong loop-free solution, there there does also not exist a relaxed solution.*

PROOF. For the sake of contradiction, assume that there exists a scenario which is solvable for RLF but not for SLF. Consider the first round in which the RLF algorithm makes the relevant update which is not possible with SLF. In this configuration, there must exist at least one backward edge and one forward edge which are not updated yet. In a scenario in which all backward edges are already updated, no forward edge can bypass the waypoint if the final policy is consistent. If all forward edges would have already been updated, then the path could easily be completed, updating the backward edges one by one starting with the backward edges connected to the forward edges.

Thus, there exists a forward edge e, which cannot be updated in SLF and by definition must be bypassing the waypoint, as forward edges in SLF cannot create loops. Since this scenario is solvable for RLF, there must exist a backward edge e' that allows the update of e at some point. As e' can only be updated for RLF but not for SLF, it must be an edge which creates a loop which is not on the path between source and destination. However, to enable the update of e, the update of e' must impact the current flow, since an update of e is only possible if the path from the head of e to the destination is leading through the waypoint. Hence, it cannot be the update of e' as it will not change the current flow, which contradicts our assumption. □

4. 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 an alternative, possibly more resource-intensive update mechanisms [4, 31]. 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 network update instance as $G(\mathcal{C})$.

Notation

In our reduction we will assume that each clause in 3-SAT has exactly 3 literals. We will denote the number of variables as k and the number of clauses as m. The variables will be denoted as $x_1, x_2, \ldots, x_k$ and the clauses as $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m$. We will 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 will also denote the clauses with literal x_i as $P_1^i, P_2^i, \ldots, P_{p_i}^i$ and the clauses with literal $\neg x_i$ as $Q_1^i, Q_2^i, \ldots, Q_{p_i}^i$.

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_1^i, d_2^i, d_3^i and for each variable we will add m nodes $r_j^1, \ldots, r_j^m$. 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.

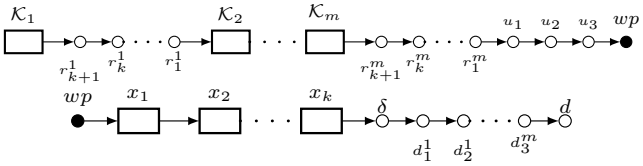

Figure 4: Order of gadgets and nodes. The upper part shows the order from the source to wp. The lower part shows the order from wp to the destination.

Variable Gadgets

For each variable x_j we construct a gadget consisting of four nodes $v_1^j, v_2^j, v_3^j, v_4^j$. These nodes are connected with the edges (v_1^j, v_3^j) and (v_2^j, v_4^j) and there is an edge from v_4^j to first node of next variable gadget, v_1^{j+1} (and in case of the last variable gadget there is an edge from v_4^k to δ). Note that these nodes v_4^j and v_1^{j+1} would be combined in our model, but it makes no difference for the problem if we keep both and the proof is easier to follow if we treat them seperately. In the gadget there are also nodes $y_1^j, y_2^j, \ldots, y_{p_i}^j$ between v_1^j and v_2^j and nodes $z_1^j, z_2^j, \ldots, z_{q_i}^j$ between v_3^j and v_4^j. Clauses $P_1^j, P_2^j, \ldots, P_{p_i}^j$ will be connected to $y_1^j, y_2^j, \ldots, y_{p_i}^j$, and updating an edge y_i^j will allow clause P_i^j to become untangled. Similarly clauses $Q_1^j, Q_2^j, \ldots, Q_{q_i}^j$ will be connected to $z_1^j, z_2^j, \ldots, z_{q_i}^j$. In turn nodes $y_1^j, y_2^j, \ldots, y_{p_i}^j$ can be updated only if v_1^j is updated, and $z_1^j, z_2^j, \ldots, z_{q_i}^j$ if v_2^j is updated and v_1^j is not updated (or all clauses are already untangled). That will allow us to conclude the value of x_j based on whether before all clauses become untangled v_1^j is updated or not. A construction of the gadget is presented in Figure 5.

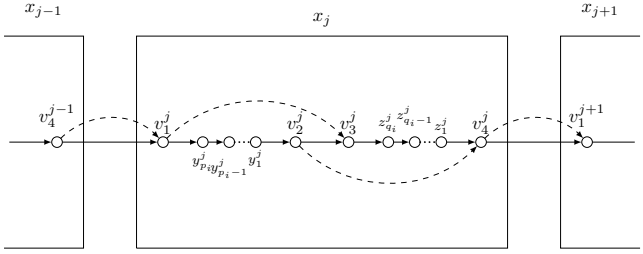

Figure 5: Construction of a variable gadget for x_j.

Clause Gadgets

For each clause $\mathcal{K}_i$ we construct a gadget consisting of nodes $c_1^i, c_2^i, \ldots, c_6^i$. Also for each clause we add three nodes (outside of the gadget, close to d, see also Figure 4) d_1^i, d_2^i, d_3^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.

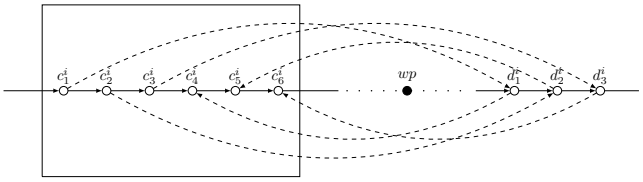

Figure 6: Construction of a clause gadget.

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 respectively 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). Note, that the nodes $y_1^i, \ldots, y_{p_i}^i$ are in reverse order than clauses $P_1^i, \ldots, P_{p_i}^i$ (so the earliest clause is connected to the last node). We proceed similarly with clauses $Q_1^i, \ldots, Q_{q_i}^i$ and nodes $z_1^i, \ldots, z_{q_i}^i$.

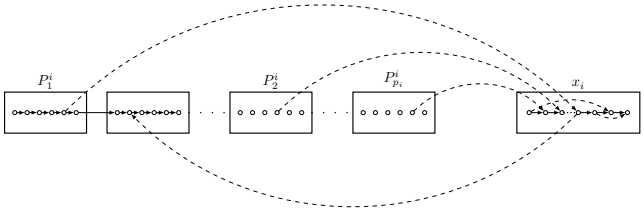

Figure 7: Edges to connect clauses

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 will be 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 i-th clause gadget we create $k+1$ nodes $r_1^i, r_2^i, \ldots, r_{k+1}^i$ in reverse order, i.e. r_{k+1}^i is the first node after the gadget and r_1^i is the last.

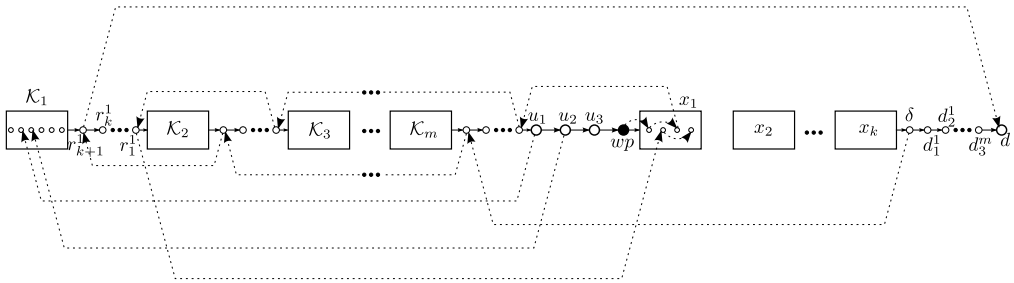

Figure 8: Connecting all paths.

For each variable x_i we create a path starting in v_3^i, then going through nodes $r_i^m, r_i^{m-1}, \ldots, r_i^1$ and ending in v_2^i. We also create a similar path starting in δ, then going through nodes $r_{k+1}^m, r_{k+1}^{m-1}, \ldots, r_{k+1}^1$ and ending in v_d^i. All these edges are shown in Figure 8.

Proof of Correctness

In this section we will prove, that the reduction is correct. We will 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 it is not untangled.

THEOREM 4. *If $\mathcal{C}$ is satisfiable then there is a schedule for $G(\mathcal{C})$ which satisfies SLF and WPE.*

PROOF. Let σ be an assignment that satisfies $\mathcal{C}$. Then based on σ we will 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. Also if $\sigma(x_i) = 1$ 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 $y_1^i, \ldots, y_{p_i}^i$, if $\sigma(x_i) = 1$, or nodes $z_1^i, \ldots, z_{q_i}^i$ otherwise.
3. Since for each clause $\mathcal{K}_j$ there is at least one literal that satisfies it, we update one of nodes c_4^j, c_5^j, c_6^j which is 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 inbetween.
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 5. *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 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. *A node r_j^i, for any $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k+1\}$, can be updated only if 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 for all $j \in \{0, 1, \ldots, m\}$ r_i^j is updated 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

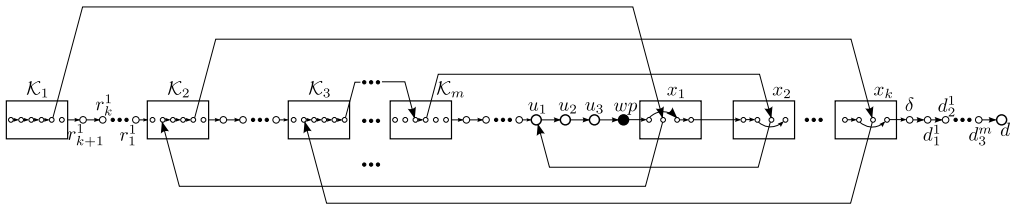

Figure 9: The path after three rounds of updating according to the schedule in proof of Theorem 4.

will go through $r_{k+1}^i, r_k^i, \ldots, r_0^i$ (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. Therefore it is enough to show that a packet enters a clause gadget twice to prove loop-freedom violation.

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 at most one condition 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_g^l to gadget $\mathcal{K}_f$. But then $f \leq i$, because of the order of nodes $y_{p_l}^l, \ldots, y_1^l$, it would go to a gadget that was already visited, and therefore violate loop-freedom. The case when the successor of c_j^i is z_a^l, for some l, a, is similar.

4. Let us assume that v_3^i is updated, but there is some r_i^j which is not updated and either v_1^i is updated or v_2^i is not. Then the packet, after going through the waypoint, reaches the gadget for x_i. Then, because v_1^i is updated or v_2^i is not, it is forwarded to v_3^i. From there it traverses through some backward edges, before it enters some clause gadget (it cannot take backward edges until it goes to r_a^1, 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_j^i is updated, then the packet traverses through all clause gadgets and variable gadgets until it reaches d_j^i. From there it goes back to c_{j+3}^i. 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_{j+3}^i is updated, the packet is forwarded to some node y_a^l (or z_a^l). 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_j^i, 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_j^i, and from there to the destination.

Finally let us consider what happens when we update v_1^i 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_1^i and next to v_3^i. From there it goes through backward edges and re-enters the clause, which violates loop-freedom. □

Now we are ready to prove Theorem 5.

Proof. Let us assume that there is a schedule for $G(\mathcal{C})$. Then let us look at the update which untangles the last clause (that is 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_a^i) and a node corresponding to negative literal (node z_l^i) that are both updated. That is because updating node y_a^i requires that v_1^i is updated, whereas updating node z_l^i requires that v_1^i is not updated. Therefore in the assignment of variables in $\mathcal{C}$ we set x_i to 1, if at least one of nodes y_a^i, for any a, is updated, or to 0 otherwise. Then because all clauses are untangled, and untangling a clause requires that at least one literal has value 1, this assignment satisfies all formulas in $\mathcal{C}$. □

Theorem 6. *There is a schedule for $G(\mathcal{C})$ satisfying RLF and WPE iff $\mathcal{C}$ is satisfiable and iff there is a schedule for $G(\mathcal{C})$ satisfying SLF and 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. Because a schedule satisfying SLF and WPE is also a schedule satisfying RLF and WPE, these three statements are equivalent. □

5. 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* [8, 21, 26, 28, 29]: sequences of network functions which are allocated and stitched together in a flexible manner. For example, a service chain c 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. Such advanced network services open an interesting new market for Internet Service Providers, which can become "miniature cloud providers" [30], specialized for in-network processing. 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 reverse order in the two policies.

THEOREM 7. *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. □

6. OPTIMAL UPDATE ALGORITHMS

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 [20] 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 adaption for realizing the relaxed loop-freedom property, 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 will refer to the MIP formulation for the decision variant of the strong loop-freedom property 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). The models and the extensions are presented in the following.

6.1 Basic Model

According to the line representation presented in Section 3, policies π_1 and π_2 are described as (simple) paths E_{π_1} and E_{π_2} on the common set of nodes V. Both E_{π_1} and E_{π_2} connect the source $s \in V$ to the destination $d \in V$.

The decision of whether switch $v \in V$ shall be updated in any 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 enforces the policy 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 $e \in E_{\pi_1} \cup E_{\pi_2}$ exists after the (successful) execution of all updates up to and including round r. Note that these variables are actually binary based on their computation as a function of the variables x_v^r. To check that the properties WPE and LF hold, transient states between the rounds $r-1$ and r need to be considered as described below.

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 the transient states between rounds to be 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 suffices to forbid cycles in the union of edges being 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, RLF only forbids loops which are reachable from the source node $s \in V$. 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 10) with corresponding leveling 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 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

Mixed-Integer Program 1: Optimal Rounds (-R-)

$$\min R \qquad \text{(Obj)}$$

$$R \geq r \cdot x_v^r \qquad r \in \mathcal{R}, v \in V \qquad (1)$$

$$1 = \textstyle\sum_{r \in \mathcal{R}} x_v^r \qquad v \in V \qquad (2)$$

$$y_{u,v}^r = 1 - \textstyle\sum_{r' \leq r} x_u^{r'} \qquad r \in \mathcal{R},\ (u,v) \in E_{\pi_1} \qquad (3)$$

$$y_{u,v}^r = \textstyle\sum_{r' \leq r} x_u^{r'} \qquad r \in \mathcal{R},\ (u,v) \in E_{\pi_2} \qquad (4)$$

$$a_s^r = 1 \qquad r \in \mathcal{R} \qquad (5)$$

$$a_v^r \geq a_u^r + y_{u,v}^{r-1} - 1 \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (6)$$

$$a_v^r \geq a_u^r + y_{u,v}^{r} - 1 \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (7)$$

$$y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^{r-1} - 1 \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (8)$$

$$y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^{r} - 1 \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (9)$$

$$y_{u,v}^{r-1 \vee r} \leq \frac{l_v^r - l_u^r - 1}{|V| - 1} + 1 \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (10)$$

$$\overline{a}_s^{r,w} = 1 \qquad r \in \mathcal{R}, w \in WP \qquad (11)$$

$$\overline{a}_v^{r,w} \geq \overline{a}_u^{r,w} + y_{u,v}^{r-1} - 1 \qquad r \in \mathcal{R}, w \in WP,\ (u,v) \in E_{\overline{\mathsf{WP}}}^w \qquad (12)$$

$$\overline{a}_v^{r,w} \geq \overline{a}_u^{r,w} + y_{u,v}^{r} - 1 \qquad r \in \mathcal{R}, w \in WP,\ (u,v) \in E_{\overline{\mathsf{WP}}}^w \qquad (13)$$

$$\overline{a}_d^{r,w} = 0 \qquad r \in \mathcal{R}, w \in WP \qquad (14)$$

the old policy E_{π_1} may be used, i.e. we set $y_{u,v}^0 = 1$ for $(u,v) \in E_{\pi_1}$ and $y_{u,v}^0 = 0$ for $(u,v) \in E_{\pi_2}$.

Enforcing WPE.

We denote by $WP \subset V$ the set of waypoints, which may under not be bypassed. 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 $w \in WP$, 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 $w \in WP$. To this end, we denote by $E_{\overline{\mathsf{WP}}}^w \subset E$ 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 $E_{\overline{\mathsf{WP}}}^w$ – no waypoint $w \in WP$ will be bypassed.

6.2 Model Extensions

Based MIP 1, we consider a series of model extensions.

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 only allows one switch to be updated within each round.

$$\sum_{v \in V} x_v^r = 1 \quad r \in \mathcal{R}\,. \qquad (15)$$

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

Enforcing SLF.

SLF is strictly stronger than RLF as it forbids cycles under any circumstances, i.e. we forbid cycles even if none of its nodes are (anymore) reachable from the source node. Hence, the reachability variables a_v^r are not needed anymore as all nodes can be considered to be reachable. Therefore, Constraints 5 - 10 can be replaced by the following constraints to enforce the SLF property.

$$y_{u,v}^{r-1 \vee r} \geq y_{u,v}^{r-1} \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (16)$$

$$y_{u,v}^{r-1 \vee r} \geq y_{u,v}^{r} \qquad r \in \mathcal{R},\ (u,v) \in E \qquad (17)$$

$$y_{u,v}^{r-1 \vee r} \leq \frac{l_v^r - l_u^r - 1}{|V| - 1} + 1 \quad r \in \mathcal{R},\ (u,v) \in E \qquad (18)$$

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 [2], which can drastically worsen the solvability in practice. To strengthen the models, we present a multi-commodity flow extension. 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 $e \in E_{\pi_1} \cup E_{\pi_2}$. Our extension can be formalized as follows:

$$\sum_{e \in \delta^+(s)} f_e^r = 1 \qquad r \in \mathcal{R} \qquad (19)$$

$$\sum_{e \in \delta^+(v)} f_e^r = \sum_{e \in \delta^-(v)} f_e^r \qquad r \in \mathcal{R}, v \in V \setminus \{s,d\} \qquad (20)$$

$$f_e^r \leq y_e^r \qquad r \in \mathcal{R}, e \in E_{\pi_1} \cup E_{\pi_2} \qquad (21)$$

$$\sum_{e \in \delta^-(w)} f_e^r \geq 1 \qquad r \in \mathcal{R}, w \in WP \qquad (22)$$

$$a_v^r \geq f_v^{r-1} \qquad r \in \mathcal{R} \qquad (23^*)$$

$$a_v^r \geq f_v^{r} \qquad r \in \mathcal{R} \qquad (24^*)$$

The first two constraints construct *s-d* flow, such that the flow starting at the source $s \in V$ must reach the destination $d \in V$. As the flow is upper bounded by the existence of the edges (see Constraint 21), 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 MIP 1. Note, that the Constraint 21 is valid both for SLF and RLF, since the flow always emerges at s and all intermediate nodes are therefore reachable.

With respect to the WPE property, the Constraint 22 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 23* and 24* 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 adaption.

7. SIMULATIONS

In our simulations we are specifically interested in the number of scenarios in which LF and WPE conflict and hence are not updateable without violating either one of the two

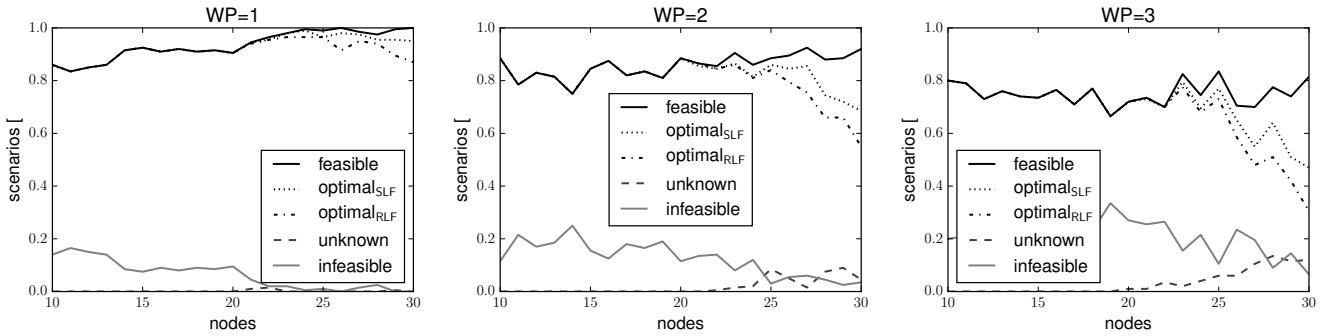

Figure 10: Classification of the generated scenarios according to whether a feasible solution exists, how many optimal solutions can be computed under strong and relaxed loop-freedom, how many scenarios are infeasible and lastly for which percentage of the scenarios neither feasibility nor infeasibility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

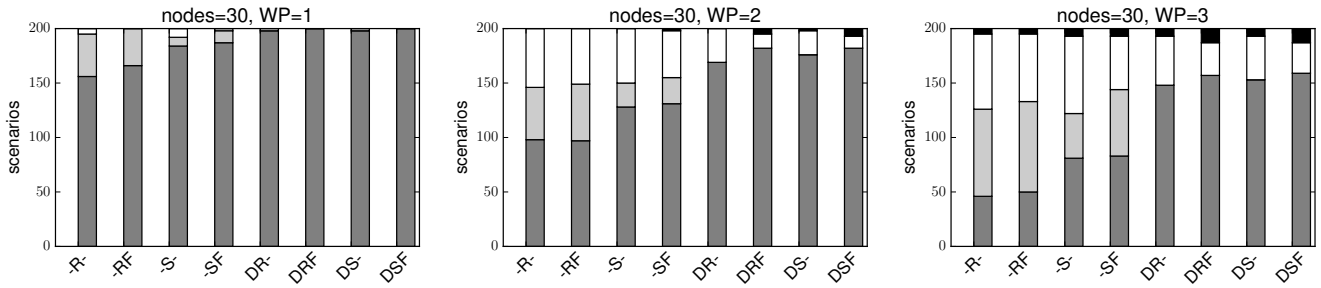

Figure 11: Qualitative evaluation of the algorithm performance for 30 switches and different number of waypoints. The bottom bar (*dark gray*) represents the (*total*) number of scenarios that can be solved optimally, the *light gray* bar represents the number of scenarios for which a feasible solution is found but optimality is not proven, the *white bars* represent the scenarios for which neither feasibility nor infeasibility can be proven in 600 seconds and the *black bars* depict the number of scenarios whose infeasibility is proven successfully.

properties. 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 by π_1 and π_2 does not differ, as we have already proven in Theorem 7 that these scenarios are unsolvable (and can easily be identified). In addition, we guarantee that no node has the same rule in π_2 as it had in π_1, as those nodes do not need to be considered within our model, since no update and hence, no changes in the forwarding behavior exists. As middleboxes are traversed one by one in the same order, the generated policies can be viewed to model service chains.

We generate updates of lengths $\{10, 11, \ldots, 30\}$, containing one to three waypoints. For each combination of these values, we generate 200 instances (12,600 overall) at random. We use Gurobi 6.5 to solve the respective eight different MIP formulations and terminate experiments, if neither optimality nor infeasibility was shown within 600 seconds.

As to be expected, the feasibility of scenarios is influenced by the number of waypoints within the policy update, see Figure 10. For a single waypoint the percentage of infeasible scenarios is decreasing when increasing the number of nodes. While roughly 15% of the scenarios are infeasible for 10 nodes, for scenarios involving at least 22 nodes only a fraction of at most 3% remain infeasible. These numbers differ when we add more waypoints. With two waypoints the number of infeasible scenarios starts out at 20% and decreases to 5%. Using three waypoints even increases the percentage initially to roughly 30% for smaller instances. 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 600 seconds, the feasibility of roughly 15% of the scenarios cannot be decided with three waypoints at 30 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 optimally.

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 30 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 for the multiple waypoint scenarios than the formulations which aim for a minimization of the number of rounds. Note that when considering the decision variant, any feasible solution is indeed optimal. 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, the optimization variants all fail to detect a significant fraction

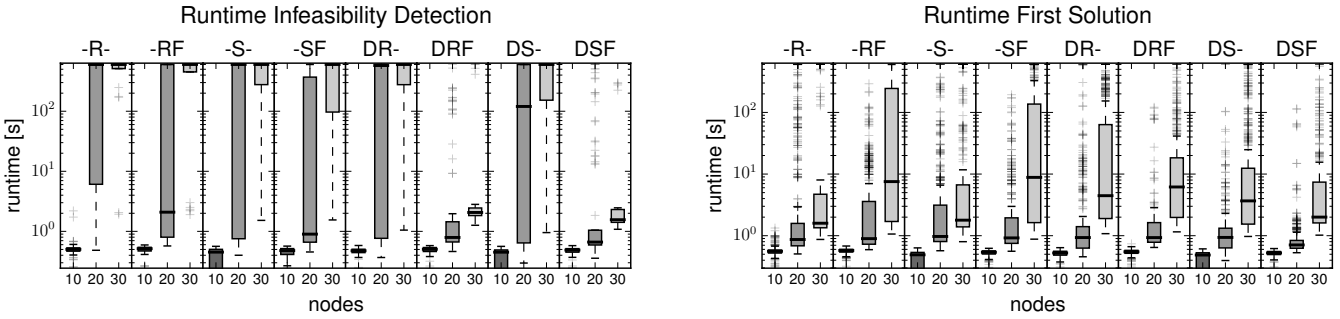

Figure 12: Runtimes for proving the infeasibility of the underlying scenario (*left*) and runtimes for finding the first feasible solution (*right*) aggregated over all number of waypoints and displayed for 10, 20, and 30 nodes as well as for each Mixed-Integer Programming formulation. Note the logarithmic y-axis.

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 very few differences on small scenario sizes, these differences increase with the scenario size. 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 all algorithms in the 20 node scenario is two orders of magnitude lower with this extension. Especially the decision variants benefit from this extension where the median is also two orders of magnitude lower in the 30 nodes scenarios. Given these results one could possibly use the decision variants as an infeasibility detector while concurrently running an instance of the minimization variant to search for applicable update schedules using the minimal number of rounds. Concretely, in contrast to the benefit for the infeasibility detection, the flow extension prolongs the time until the first solution is found for the minimization algorithms and has roughly no impact in the decision variants. The plots of the runtimes for finding the optimal solution look similar to those for finding the first solution and we hence omit it here.

As the overall goal is to minimize the number of rounds, we lastly present in Figure 13 an ECDF for comparing the minimal number of rounds required to update the different scenarios. We distinguish between RLF and SLF consider scenarios with 10, 20, and 30 nodes. We observe that more than 90% of the scenarios are solvable with at most 10 rounds. Furthermore, the usage of RLF decreases the required number of rounds by roughly one on average.

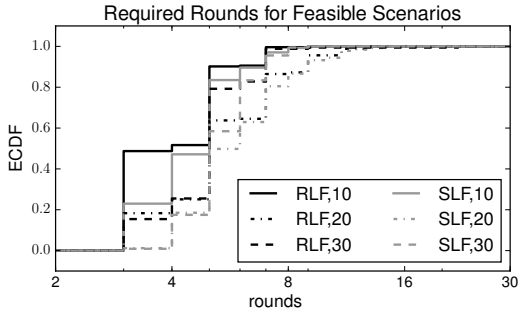

Figure 13: ECDF of the minimal number of required rounds for updating (*feasible*) scenarios with 10 (solid), 20 (dash-dotted), and 30 (dashed) nodes under relaxed (*black*) and strong loop-freedom (*gray*). More than 90% of the scenarios can be solved using 10 or less rounds. Note the logarithmic x-axis.

8. RELATED WORK

Networking routing comes with many policy constraints today, and the impact of these constraints and how to take them into account algorithmically has been studied intensively over the last years, specially in the context of the wide-area Internet [3, 16, 29, 32]. We in this paper are particularly interested in routes maintaining waypoint enforcement guarantees.

The problem of updating [4, 10, 13, 22, 31], synthesizing [24] and checking [15] SDN policies [25] as well as routes [5] has also been studied intensively. In their seminal work, Reitblatt et al. [31] initiated the study of network updates providing strong, per-packet consistency guarantees, and the authors also presented a 2-phase commit protocol. This protocol also forms the basis of the distributed control plane implementation in [4]. Mahajan and Wattenhofer [22] started investigating weaker transient consistency properties—in particular also (strong) loop-freedom—for destination-based routing policies. The measurement studies in [13] and [18] provide empirical evidence for the non-negligible time and high variance of switch updates, further motivating their and our work. In their paper, Mahajan and Wattenhofer proposed an algorithm to "greedily" select a maximum number of edges which can be used early during the policy installation process. Our work builds upon [22], and we consider the scheduling complexity of updating arbitrary routes which are not necessarily destination or shortest-path based. The weak-consistency model introduced in [22] already led to several follow-up papers, for example [19] which also studies round-based models and [7] which studies extensions to jointly optimizing multiple policies. There have recently also been proposals to prioritize shorter policies over longer policies, in order to improve the overall update time [27]. Such a prioritization is meaningful and orthogonal (i.e., could be used in addition) to our approach which optimizes a single policy.

Researchers have also started investigating consistent updates for networks which include (network function virtualized) middleboxes [23]. In their interesting work [9], Ghorbani and Godfrey argue that in the context of network function virtualization, rather stronger consistency properties are required.

A workshop version of this paper appeared at HotNets 2014 [20].

9. CONCLUSION

While software-defined and network function virtualized networks introduce a more flexible and dynamic network operation, this new paradigm also introduces fundamental new challenges. This paper initiated the study of how to update a network in a transiently consistent manner, ensuring two most basic invariants, loop-freedom and waypoint enforcement. We first made several fundamental observations about this problem, and then presented a rigorous proof that it is NP-hard to decide whether a feasible update schedule satisfying both invariants exist. We complemented this negative result with multiple exact algorithms which not only allow to quickly, in practice, identify infeasible instanced (in which case one could resort to update schemes based on tagging), but also to compute optimal transiently consistent update schedules, whenever they exist. We also initiated the discussion of scenarios in which network functions need to be traversed in a certain order, and complemented our formal results with an extensive simulation study. We believe that our paper opens a rich and interesting area of research. Indeed, the required level of consistency often depends on the specific setting, and we will further explore the different notions of consistent updates in the presence of network functionality.

Acknowledgments. This research was supported by the EU project UNIFY FP7-IP-619609 and a German BMBF Software Campus grant (01IS12056).

10. REFERENCES

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

[2] D. Bertsimas and R. Weismantel. *Optimization over integers*, volume 13. Dynamic Ideas Belmont, 2005.

[3] M. Caesar and J. Rexford. Bgp routing policies in isp networks. *Netwrk. Mag. of Global Internetwkg.*, 19(6):5–11, Nov. 2005.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. USENIX NSDI*, 2013.

[16] R. Klöti, V. Kotronis, B. Ager, and X. Dimitropoulos. Policy-compliant path diversity and bisection bandwidth. In *Proc. IEEE INFOCOM*, 2015.

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

[18] 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.

[19] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.

[20] 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.

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

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

[23] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *Proc. HotSDN*, pages 67–72, 2013.

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

[25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[26] P. Skoldstrom et al. Towards unified programmability of cloud and carrier infrastructure. In *Proc. European Workshop on Software Defined Networking (EWSDN)*, 2014.

[27] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostić. Espres: transparent sdn update scheduling. In *Proc. ACM HotSDN*, pages 73–78. ACM, 2014.

[28] R. Hartert et al. Declarative and expressive approach to control forwarding paths in carrier-grade networks. In *Proc. ACM SIGCOMM*, 2015.

[29] R. Soulé et al. Merlin: A language for provisioning network resources. In *Proc. 10th ACM CoNEXT*, 2014.

[30] R. Stoenescu et al. In-net: Enabling in-network processing for the masses. In *Proc. ACM EuroSys*, 2015.

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

[32] L. Sobrinho and T. Quelhas. A theory for the connectivity discovered by routing protocols. In *Proc. IEEE/ACM Trans. Netw.*, 2012.