Safe, Efficient and Robust SDN Updates by Combining Rule Replacements and Additions

Stefano Vissicchio, Luca Cittadini

***Abstract*—Disruption-free updates are a key primitive to effectively operate SDN networks and maximize the benefits of their programmability. In this paper, we study how to implement this primitive safely (with respect to forwarding correctness and policies), efficiently (in terms of consumed network resources) and robustly to unpredictable factors like delayed message delivery and processing. First, we analyze the fundamental limitations of prior proposals, which either (i) progressively replace initial flow rules with new ones, or (ii) instruct switches to maintain both initial and final rules. Second, we show that safe, efficient and robust updates can be achieved by leveraging a more general approach. We indeed unveil a dualism between rule replacements and additions, that opens new degrees of freedom for supporting SDN updates. Third, we demonstrate how to build upon this dualism. We propose FLIP, an algorithm that computes operational sequences combining the efficiency of rule replacements with the applicability of rule additions. FLIP identifies constraints on rule replacements and additions that independently prevent safety violations from occurring during the update. Then, it explores the solution space by swapping constraints that prevent the same safety violations, until it reaches a satisfiable set of constraints. Fourth, we perform extensive simulations, showing that FLIP can significantly outperform prior work: In the average case, it guarantees a much higher success rate than algorithms only based on rule replacements, and massively reduces the memory overhead needed by techniques solely using rule additions.**

I. Introduction and Related Work

The single most important function of an SDN controller is deciding how packets are forwarded through the network, and program the switches accordingly. Updates are often needed to adapt forwarding paths to network dynamics, e.g., to better balance load, steer flows through virtualized functions or implement new security policies. During an update, the controller has to instruct switches to add, change and remove some of the flow rules that they use to forward packets.

Ideally, the controller should carry out any update in a safe, efficient and robust way. By safety we mean that service disruptions should be avoided, hence both forwarding correctness (i.e., packet delivery) and policies (i.e., requirements on forwarding paths) have to be preserved throughout the update. In addition, the update should be efficient in terms of consumed network resources (from bandwidth to switch memory). Finally, the update strategy should be robust to factors unpredictable a-priori, like non-deterministic processing time for switches to install or modify rules, or (indefinitely) delayed message delivery between the controller and the switches. This robustness requirement rules out the naive approach of pushing the final rules to all switches at the same time, as well as strategies based on simultaneously applying operations on several switches [23]. Rather, the controller must apply a carefully-computed operational sequence, so that it can either perform the next operation or roll back the previous one, at any time, while provably preserving the update safety.

Despite the abundant literature on SDN updates (see [6] for an overview), no proposed techniques supports safe, policy-preserving updates, in an efficient and robust way.

- Many proposals focus on congestion avoidance [3], [7], [9], [15] or forwarding correctness [17], [19], [32], but do not support policy preservation at all.
- Some techniques [18], [20] support policies by computing a specific order to replace initial rules with final ones. We will refer to them as *ordered replacement* techniques. They are efficient but their applicability is limited: An order that guarantees both forwarding and policy preservation may not exist [18], and it is computationally hard to even decide if such an order exists [16].
- Finally, *two-phase commit* techniques [12], [24], [28] instruct network devices to temporarily store the initial and final version of every rule that has to be updated. Either initial or final rules are applied consistently network-wide, depending on tags that are explicitly set on each packet at the ingress (like in [12], [28]) or implicitly inferred by switches (e.g., on the basis of packet timestamps [24]). While natively preserving forwarding correctness and policies, this approach is inefficient, up to the point of being impractical [12], [21].[1]

In particular, two-phase commit techniques inefficiently use device memory, which is a precious resource, so important that its consumption is regarded as one of the key factors for scalable routing systems [36]. Device memory is a primary concern for current SDN networks, since commercial SDN switches employ Ternary Content Addressable Memory (TCAM) to support programmability [22]: TCAM is expensive, power-hungry [14], and scarce [34]. We expect that memory consumption keep being an important concern in the future, e.g., considering that IPv6 requires more bytes per flow than IPv4. Even more fundamentally, device memory is a resource that must be shared among all packet-processing network applications. Instead of reserving memory for network updates, operators might therefore need to use that memory to support the always growing number of offered services,

S. Vissicchio is a lecturer at University College London. He developed most of this work when he was postdoctoral researcher of F.R.S.-FNRS, at Université catholique de Louvain. E-mail: s.vissicchio@cs.ucl.ac.uk
L. Cittadini is with Roma Tre University. E-mail: luca.cittadini@gmail.com
This work has been partially supported by ARC grant 13/18-054 from Communauté française de Belgique. A preliminary version of this paper appeared in [31].

[1] Note that works like [10] building upon two-phase commit techniques (to guarantee higher-level properties) also inherit this limitation.

and to guarantee good network performance – for example, supporting fine-grained (i) traffic engineering [11], (ii) security and monitoring tasks [25], or (iii) fast failure recovery [27].

In this paper, we propose a model and an algorithm to compute operational sequences that preserve forwarding correctness and policies, using additional rules only if necessary. Our contributions are complementary to works that optimize the implementation of rule replacement and additions (e.g., by avoiding unnecessary updates of rule priorities [34]).

We unveil the dualism between rule replacements and additions, showing that forwarding disruptions and policy violations can be prevented by either adding rules or constraining the order of their replacement. This dualism allows us to explore the solution space with new degrees of freedom.

We show that combining replacements and additions is more powerful than restricting to either of the two, as all previous techniques do. Such combinations, indeed, enable new ways to guarantee the update safety, e.g., by admitting harmless forwarding loops that packets traverse exactly once before being successfully forwarded to their destination.

Unsurprisingly, this additional expressiveness comes at a cost: It makes the problem of finding a safe update sequence more challenging. Indeed, it significantly increases the search space, since many more solutions are possible (all combinations of rule replacements and additions). Moreover, it requires a deeper understanding of the interactions between rule replacements and additions performed on different switches, e.g., distinguishing (at computation time) loops that are crossed only once from those that disrupt connectivity.

We address those challenges with an original algorithm, called FLIP. To compactly represent the search space, FLIP formalizes possibilities to avoid safety violations as constraints on rule replacements and additions. Moreover, it discovers relationships between those constraints: It identifies sets of constraints that are alternative to each other, as they are capable of preventing the same forwarding disruption or the same policy violation. For example, given a potential policy violation, FLIP can determine that either constraints A and B must be enforced for certain rule replacements, or constraint C must hold for a given rule addition. FLIP then explores the search space by swapping constraints with their alternatives, until it ends up with a satisfiable set of constraints.

FLIP supports safe updates that cannot be carried out with only rule replacements or solely rule additions. Moreover, it greatly reduces the number of added rules in the average case. When combining replacements and additions is not advantageous for safety or efficiency, FLIP degenerates to either ordered replacement or two-phase commit. This guarantees that: (i) FLIP always computes a zero-overhead sequence, if one exists (as ordered replacement); and (ii) FLIP always finds a solution whenever any of the previous techniques is applicable, e.g., if all network nodes have space to install one additional rule (as two-phase commit). In those cases, though, FLIP inherits the limits of the approach to which it degenerates. For example, it induces the same overhead as two-phase commit techniques if it is not safe to perform any rule replacement; also, FLIP may not support some update scenarios if rules cannot be added on any network node.

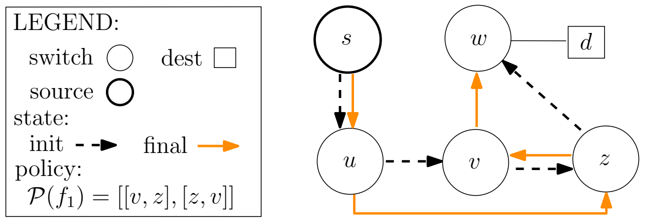


Fig. 1. An update scenario with a policy to be preserved.

The rest of the paper describes the following contributions.

Analysis (§II). We detail how combining rule replacements and additions opens new degrees of freedom in the policy-preserving update problem. Also, we show how those combinations enable us to overcome limitations of prior techniques.

Modeling (§III). We formalize the safe update problem when operational sequences can include rule replacements and additions. We also describe how FLIP models the solution space in terms of constraints and relationships between them.

Algorithms (§IV-V). We walk through the execution of FLIP, and detail its core procedures to extract constraints, identify their relationships, and compute safe operational sequences.

Experimental evaluation (§VI). We evaluate our implementation of FLIP by simulating $50,000$ random update scenarios for realistic networks: FLIP systematically outperforms previous techniques in terms of efficiency and success rate.

II. Unexplored Degrees of Freedom for SDN Updates

Fig. 1 shows a case where the SDN controller (not depicted to help the reader focus on forwarding paths) has to update the controlled network. For the sake of the example, the controller has to modify the forwarding only for the flow f_1 of packets sourced at s and destined to d. Dashed and solid arrows respectively represent the initial and final states, i.e., the paths used before and after the update.

To perform the update, the controller can apply atomic operations on switches. Specifically, it can add, modify or delete the flow rules used by any switch to process packets belonging to f_1. We distinguish three types of operations, readily supported by SDN switches. A *rule replacement* operation $rep(s,f)$ instructs a switch s to replace all its current rules for any flow f with the final rule. A *tagging* operation $tag(s,f,\theta)$ requires switch s to mark packets in flow f with a tag θ. A *matching* operation $match(s,f,\theta_i,\theta_f)$ requests switch s to install both the initial and final rules for flow f, and apply the initial (final, resp.) rule to packets tagged as θ_i (θ_f, resp.). In our notation, $\emptyset$ is a valid value for any tag, and represents the absence of a tag. Both rule replacement and tagging operations modify an existing rule, hence they do not change the number of installed rules. Conversely, a matching operation involves adding a new rule, and consumes an additional slot in the TCAM memory of the corresponding switch. We denote with $app(op)$ the time at which operation op is applied.

We say that the controller produces a *safe update* if (i) packets are guaranteedly delivered to d; and (ii) input policies are satisfied throughout the update. In our example, the policy $\mathcal{P}(f_1)$ (see left side of Fig. 1) imposes that packets belonging to f_1 must traverse link (v, z) in either of the two directions. We obviously assume that properties (i) and (ii) hold in the initial and final states (otherwise no update can be safe).

Update safety depends on the sequence of operations applied to the switches. In Fig. 1, for instance, if the first operation is replacing the rule on z, i.e., $rep(z, f_1)$, then packets for flow f_1 are trapped in a permanent loop between v (that applies its initial rule) and z (that applies its final rule) after $app(rep(z, f_1))$. The loop persists until $app(rep(v, f_1))$. Instead, if $rep(u, f_1)$ is the first operation, then f_1 is forwarded over path $[s, u, z, w, d]$, hence violating the policy $\mathcal{P}(f_1)$.

A. *Previous approaches have limitations*

To achieve safe updates, prior work either relies on ordered replacements or on two-phase commit. The former approach consists in computing a proper sequence of rule replacements, when it exists (see, e.g., [18], [20]). The latter one works in two phases: In the first phase, it applies matching operations on all internal switches (u, v and z in Fig. 1), in the second phase it applies tagging operations on flow entry points (s in Fig. 1) so that all switches use final rules (see, e.g., [12], [28]).

Both approaches are limited in applicability or inefficiency, since they focus either only on rule replacements or exclusively on tagging and matching operations.

Ordered replacement cannot always be applied. Fig. 1 proves that an ordering of rule replacements preserving both forwarding correctness and given policies ($P(f_1)$ in this case) does not always exist. Consider possible orderings of rule replacements at u, v, and z. We have three cases. If we start from u and $rep(u, f_1)$ is the first operation to be applied by the controller, then f_1 is forwarded on path $[s, u, z, w, d]$ upon $app(rep(u, f_1))$, which immediately violates $\mathcal{P}(f_1)$. If we start from v, then f_1 is forwarded on path $[s, u, v, w, d]$ upon $app(rep(v, f_1))$, which also violates $\mathcal{P}(f_1)$. Finally, if $rep(z, f_1)$ is the first operation, packets of f_1 are trapped in a permanent loop between v and z.

Simultaneous operations are not robust to unpredictable delays. One may be tempted to impose that some rules are replaced simultaneously [23], for example on u, v and z. Unfortunately, it is practically impossible to ensure that those replacements are actually executed at the same time on the respective switches. For example, z may be slower (by seconds [10]) than u to replace its initial rule with its final one, which would lead to the violation of $\mathcal{P}(f_1)$. Further, v can keep using its initial rules for an undefined time, e.g., because of a lost message in the communication between the controller and v itself: This would trigger the loop between v and z (potentially, even after the previous policy violation). Hence, simultaneous operations provide no guarantees on update safety in practice. On the contrary, they can cause all the forwarding and policy violations (potentially, one after the other) raised by unsafe rule-replacement orderings.

Two-phase commit techniques are inefficient. They are based on applying tagging and matching operations on internal switches in the network. This comes with two possible consequences. First, the technique is not applicable if there are switches (say u) that cannot accept additional rules – e.g., if their memory is fully used by rules generated (possibly at runtime) for (i) dynamic load-balancing and fine-grained traffic engineering [11], (ii) detailed monitoring and troubleshooting for security tasks [25], or (iii) fast failure reaction through pre-provisioned backup rules [27]. Second, even if the technique is applicable, it can require an unnecessarily high number of additional rules, consuming memory resources on the switches and potentially inhibiting other applications as dynamic traffic engineering or fine-grained monitoring during the update.

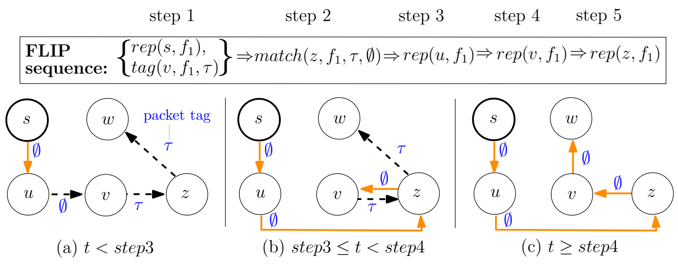

step 1
step 2
step 3
step 4
step 5
FLIP sequence:
$\left\{\begin{matrix} rep(s, f_1), \\ tag(v, f_1, \tau) \end{matrix}\right\} \Rightarrow match(z, f_1, \tau, \emptyset) \Rightarrow rep(u, f_1) \Rightarrow rep(v, f_1) \Rightarrow rep(z, f_1)$
packet tag
s
w
u
v
z
$\emptyset$
τ
(a) $t < step3$
(b) $step3 \leq t < step4$
(c) $t \geq step4$

Fig. 2. FLIP operational sequence for the scenario in Fig. 1: The overhead is only one additional rule (due to matching operation on z) versus the three additional rules needed by two-phase commit techniques like [28].

B. *Combining operations is more powerful*

The key intuition exploited by our algorithm, FLIP, is that we can profitably combine rule replacement, tagging and matching operations. To this end, we build upon properties that hold if given operations are applied in a certain order. In Fig. 1, for instance, matching on z ensures that the (v, z) link is traversed at least once, while tagging on v with z matching v's tags ensures that packets exit the potential loop between v and z after traversing z at most twice. From those guarantees, we can compute a safe update (e.g., with v tagging and z matching v's tags throughout the process).

The exact operational sequence computed by FLIP on the scenario in Fig. 1 is reported at the top of Fig. 2. It consists of a sequence of update steps, so that operations in one step have to be applied after those in the previous step. This means that the controller must start the operations in a step only after it is sure that all the operations in the previous step are applied (e.g., after receiving an acknowledgment from switches [13]). Operations in the same step can be sent simultaneously to the switches. This does not mean that they are executed simultaneously, rather that their relative order does not matter.

FLIP admits correct paths impossible in other approaches. The bottom part of Fig. 2 provides an illustration of the paths followed by packets of f_1 in *any possible state* derived from the application of the FLIP sequence. It visually proves that both packet delivery and policy compliance (i.e., traversal of the link between v and z) are guaranteed. Indeed, packets either follow the initial paths (see Fig. 2(a)), are forwarded over final paths (see Fig. 2(c)), or traverse link (v, z) in both directions before exiting the loop between v and z after one lap (see Fig. 2(b)). Note that the intermediate state in Fig. 2(b) can only be setup by conveniently interleaving rule replacements and additions, hence it is structurally impossible for both ordered replacement and two-phase commit techniques (including those relying on time-based tags like [24]).

This ability of installing additional paths leads to the following key benefits with respect to previous approaches.

FLIP can carry out updates that are not supported by any previous technique. For example, if the memory of u is fully used, Fig. 1 shows a scenario solved by FLIP, while neither ordered replacement nor two-phase commit techniques can be used (see discussion in §II-A). Contrary to time-dependent techniques [23], FLIP updates are robust to unpredictable delays in message delivery and operation application. The controller only has to check that all the operations in one step have been correctly applied before starting the next step. Indeed, operations in each single step can be applied safely irrespective of their relative order.

FLIP is more efficient than two-phase commit. Consider again the example in Fig. 1. FLIP's overhead is a single additional rule on z. This overhead is much less than the one of two-phase commit techniques, as the latter ones (when they can be used) would install additional rules on u, v, and z.

One may argue that two-phase commit techniques have been introduced to support strong consistency [6], i.e., to guarantee that only the initial or the final paths are used throughout an update, for every flow. Not only does FLIP support strong consistency too, but it also uses fewer rules than two-phase commit techniques to provide such support. In Fig. 1, for instance, FLIP adds rules only on v and z. Indeed, packets in f_1 are forwarded on either the initial or the final path if v and z apply the initial or final rule consistently with u. FLIP ensures this property by (i) instructing u to add a tag θ_f when it uses its final rule, and (ii) forcing v and z to apply their final rules when matching θ_f.

FLIP's gains tend to be even bigger with a higher number of flows to update. Table I details how previous techniques and FLIP perform when u has a limited number of free memory slots and rules for N flows have to be updated as in Fig. 1.

Ordered replacement techniques are still not applicable. The original two-phase commit algorithm [28] cannot handle updates unless u has at least N free slots. In that case, it completes the update in only 3 steps (i.e., installing matching rules on internal switches, instructing border switches to tag packets, and removing old rules). However, it is even more memory inefficient than in the single-flow case: It consumes exactly N entries on u, v and z, for a total of 3N rules versus the N rules added by FLIP (e.g., on z only). A workaround [12] to support safe updates of multiple flows when u has X<N free slots is to shard the update in rounds, so that at most X flows are updated in each round. This dilutes the memory overhead over time: the total number of rules added during the update remains 3N, but each round adds only a fraction of them (at most 3X). Such workaround, however, has a detrimental impact on the speed of the update: Each round is performed in 3 steps (as for the original two-phase commit algorithm), with each step realistically taking order of seconds. Indeed, to complete any update step, environmental factors (like propagation delay between the SDN controller and switches, rule installation, reception of acknowledgements, and error recovery) can easily account for hundreds of milliseconds in a geographically-distributed network. Even more importantly, recent works (e.g., [10], [35]) have shown that any current OpenFlow device can take a few seconds to install a hundred rules. As a final result, sharding an update in a few tens (hundreds, resp.) steps increases the update time from order of seconds to order of minutes (hours, resp.). By reducing the number of added rules, FLIP achieves a superior memory-time trade-off (e.g., see Table I).

	Free memory slots on u		
	N/100	N/10	N
ordered replacement	-	-	-
two-phase commit (base)			
- update steps	-	-	3
- estimated update time*	-	-	seconds
- total number of added rules			3N
two-phase commit (progressive)			
- update steps	300	30	3
- estimated update time*	tens of minutes	minutes	seconds
- total number of added rules	3N	3N	3N
FLIP (this paper)			
- update steps	5	5	5
- estimated update time*	seconds	seconds	seconds
- total number of added rules	N	N	N

* based on performance of current SDN switches (e.g., see [10])

TABLE I
FEATURES OF UPDATES COMPUTED BY DIFFERENT TECHNIQUES, WHEN PATHS FOR $N \geq 100$ FLOWS HAVE TO BE CHANGED AS IN FIG. 1.

Table I remains exactly the same if v or z are also memory constrained: In the first case, FLIP would still compute the sequence shown in Fig. 2; in the latter case, it would compute a symmetric update sequence where z tags and v matches.

We finally note that the percentage of rule saved by FLIP with respect to two-phase techniques is much higher in our experiments (90-98%) than the one shown in the table (66%).

III. Systematically Combining Rule Replacements and Additions

Fig. 3 overviews FLIP. We now describe FLIP's input (§III-A), output (§III-B), and algorithmic core (§III-C). Since we publicly released our FLIP implementation [31], we omit its formalization (i.e., pseudo-code) and provide a plain-text description. We use the terms switch and node interchangeably.

A. FLIP Input

FLIP takes as input an *update problem*, which is defined by the pair of initial and final states, and the properties that have to preserved during the update.

Initial and final states are defined by per-flow rules used by switches before and after the update, respectively. We consider the concept of *flow* in its broadest sense, as the collection of all

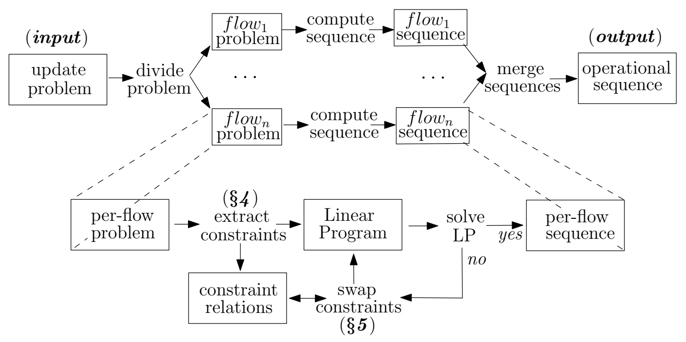


Fig. 3. High-level view of FLIP algorithm. Non-boxed text are used for FLIP internal procedures, and text boxes for the corresponding input and output.

packets whose headers match a specific bitmask consistently across switches. In Fig. 1, all switches match packets based on a bitmask that captures the source address s and the destination address d; hence, packets sourced at s and destined to d belong to the same flow f_1. Each flow is associated to a destination to which packets have to be delivered and a set of sources, e.g., switches attached to the origin of the packets. We define the *forwarding paths* for a flow f as the network paths $[s_0, s_1, s_2, \dots, d]$, where s_0 is a source, each s_i is a switch, and d is the destination. We admit multiple forwarding paths (e.g., equal-cost multipath, ECMP), between a source and a destination for the same flow.

Properties to be guaranteed include forwarding correctness and preservation of input policies.

Forwarding correctness means that every packet is delivered to the destination. Even assuming that the initial and final states are forwarding correct, two types of incorrectness can be triggered in intermediate states, installed during the update: blackholes and evil loops. A *blackhole* occurs when a forwarding path $[s, \dots, b]$ terminates in a switch b, different from the destination and without a rule to forward the packet further. An *evil loop* occurs when packets of a given flow are bounced back and forth indefinitely, among a finite number of switches. In other words, evil loops make a forwarding path infinite. Note that the loop in Fig. 2(b) is not evil since the forwarding path used for f_1, i.e., $[s, u, z, v, z, w, d]$, is finite. In the following we use the term loop to indicate an evil loop occurring during the update, unless otherwise specified.

Policy preservation means that a set of input policies, satisfied in both the initial and final states, are not violated in any intermediate state. With respect to previous works that either support strong consistency [10], [28] or single-node traversal [18], FLIP can preserve a larger variety of practical policies. Policies supported by FLIP include traversal of single nodes or links (e.g., for firewalling [28]), but also of sub-paths (e.g., for distributed middleboxing [26], service chaining [8] or QoS-based traffic engineering [1]). Generalizing the notation in Fig. 1, we define a policy as a set of non-empty paths, called *policy paths*. An input policy $\mathcal{P}(\{f_1, \dots, f_k\}) = [P_1, \dots, P_m]$, with $k, m \geq 1$, imposes that every forwarding path of any flow f_i, with $i = 1, \dots, k$, includes at least one of the policy paths $P_1, \dots, P_m$. If this condition holds, we say that the policy is *satisfied*; otherwise, we say that it is *violated*. We assume that only one policy is defined for any flow. This, however, does not prevent us from forcing the same flow through multiple sub-paths (e.g., for service chaining). For example, if a given flow has to traverse both sub-paths P_1 and P_2, we can express this requirement with a single policy including all paths $P_1 Q_i P_2$, where Q_i is a path between P_1 and P_2.

B. FLIP Output

FLIP returns a partial order between operations. This partial order represents an *operational sequence*, including rule replacement, tagging and matching operations. A returned sequence $[G_1, \dots, G_n]$ is such that (i) every G_i, with $i = 1, \dots, n$, is a group of operations; (ii) operations in each group G_i guarantee preservation of forwarding correctness and input policies independently of the relative order in which they are actually applied by switches (i.e., they can be sent by the controller in any order or in parallel); and (iii) no operation of a group G_{i+1} should be executed before any operation in G_i. We refer to any group G_j as j-th *update step*.

The above properties of FLIP sequences guarantee maximum robustness to uncontrollable factors: The resulting updates are indeed safe even if any message between the controller and switches is subject to an arbitrary large but finite delay (e.g., they will be retransmitted if lost), and any switch takes a non-deterministic time [10] to apply an operation after receiving the corresponding message from the controller. This is because all intermediate states are safe irrespectively of the execution order of operations in the same step – see property (ii). Hence, the controller can always pause the update for an arbitrary amount of time, or even roll-back to a previous state (undoing all the operations in the current update step).

C. Algorithmic Overview

At a high-level, FLIP adopts a *divide-and-conquer* approach (see top of Fig. 3). It divides the input update problem into sub-problems, one per impacted flow. For every sub-problem, FLIP independently computes a sequence. Per-flow sequences are finally merged into the output operational sequence.

Problem decomposition and solution composition are easy. Flows are by definition independent of each other, so we decompose the problem by simply considering one flow at a time. For the same reason, per-flow sequences can be arbitrarily merged without impacting forwarding correctness and policy preservation. FLIP relies on a simple yet generic strategy in which per-flow sequences are merged on a per-step basis. Starting from a set of per-flow sequences, FLIP computes the i-th step of the final operational sequence as the union of the i-th step of all per-flow sequences with at least i steps. This implies that the final sequence is as long as the longest per-flow sequence. Note that more sophisticated merging strategies are possible. For example, we could treat each per-flow sequences as a set of dependencies and use a scheduling algorithm in [10] to optimize the update speed.

Computation of policy-preserving per-flow sequences is the most novel part of FLIP. It is based on two core procedures, which are detailed in the following two sections (§IV-V).

The **constraint extraction** procedure takes as input a per-flow problem and performs two tasks.

First, for each possible forwarding incorrectness or policy violation, the procedure *identifies the constraints* that (if satisfied) ensure a safe update. We distinguish between replacement and tag-and-match constraints. A *replacement constraint* imposes a certain ordering between rule replacements. A *tag-and-match constraint* forces some switches to tag packets consistently with the applied rule (initial or final), and other switches to match those tags. For example, to avoid the loop between v and z in Fig. 1, the replacement constraint generated by FLIP is $app(rep(v, f_1)) < app(rep(z, f_1))$. The tag-and-match constraint for the same loop imposes that v tags and z matches until all the switches use their final rules. To setup packet tagging and tag matching, the latter constraint requires that $tag(v, f_1, \tau)$ and $match(z, f_1, \tau, \emptyset)$ are respectively in G_1 and G_2, i.e., the first and second update steps. Also, to force z to match throughout the update, the constraint mandates $app(rep(z, f_1)) > app(rep(n, f_1))$ for any switch $n \neq z$.

Second, the constraint extraction procedure infers *relationships between constraints*. Namely, it pinpoints alternative and dependent constraints. A set of constraints A is *alternative* to another set of constraints B if satisfying A prevents all the potential correctness violations that would be prevented by satisfying B. For example, applying a rule replacement on v before z, applying a matching operation on z (with v tagging), and applying a matching operation on v (while z tags) are all alternative constraints to avoid the evil loop between z and v in Fig. 1. In contrast, one constraint c_1 *depends* on another constraint c_2 if every time we want to impose c_1 we must also impose c_2. We will discuss dependencies in more detail in §V.

After having extracted constraints, FLIP selects all rule replacement constraints and marks them as *active*. FLIP then translates the set of active constraints into a linear program (LP) where the objective function is to minimize the number of update steps. FLIP tries to solve this LP with standard optimization algorithms. If a solution can be found, FLIP outputs the corresponding operational sequence. Otherwise, FLIP applies the **constraint swapping** procedure to replace some active constraints with alternative ones and their dependencies. Whenever a rule can be added to all switches, matching constraints are always satisfiable, hence FLIP eventually reaches a combination of active constraints for which a solution exists.

IV. FLIP Constraint Extraction

We now describe the constraint extraction procedure, using Fig. 4 for illustration.

We start by defining the concept of *crucial predecessors*, which is used in the entire procedure. Intuitively, crucial predecessors of node n are those predecessors of n that can interrupt an initial or final forwarding path traversing n, depending on whether they are updated or not. More precisely, given a node n, a flow f, and a state σ which is either the initial state or the final state, i.e., $\sigma \in \{init, fin\}$, we define crucial predecessors of n for f in σ a set C of nodes such that for every forwarding path $Q = [s \dots n \dots d]$ in σ, Q can be written as $[s \dots p, m \dots n \dots d]$, with possibly $m = n$,

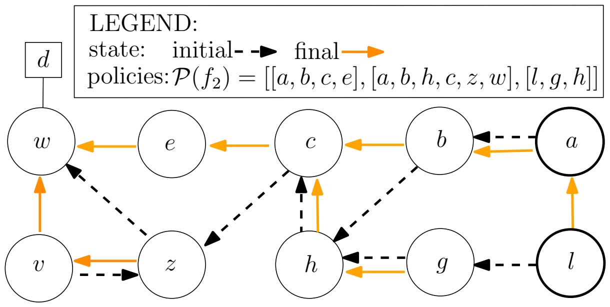

LEGEND:
state: initial
final
policies: $\mathcal{P}(f_2) = [[a, b, c, e], [a, b, h, c, z, w], [l, g, h]]$
d
w
e
c
b
a
v
z
h
g
l

Fig. 4. Update scenario used to illustrate FLIP constraint extraction.

$p \in C$, possibly $p = s$, and m next-hop of p only in σ (but not in $\{init, fin\} \setminus \{\sigma\}$). Crucial predecessors are initial if $\sigma = init$, and final otherwise. In Fig. 4, a set of initial crucial predecessors of w for the considered flow f_2 is $\{z\}$. Indeed, initial source-destination paths $[x \dots w, d]$, with $x \in \{l, a\}$, can all be rewritten as $[x \dots z, w, d]$ and w is not the next-hop of z in the final state. A node can have multiple sets of crucial predecessors. For example, $\{z\}$ and $\{c\}$ are two distinct sets of initial crucial predecessors of w for f_2 in Fig. 4. Whenever this case holds, we always consider a specific set of crucial predecessors which we denote as $cpreds(n, f, \sigma)$. This set has the additional property that for every forwarding path $Q = [s \dots p \dots n \dots d]$, with $p \in cpreds(n, f, \sigma)$, every node in the sub-path of Q from p to n (if any) uses the same next-hop in both the initial and the final states for f. As a result, $cpreds(w, f_2, init) = \{z\}$ in Fig. 4, because the subpath from z to w does not contain any intermediate node. On the other hand, the subpath from c to w contains z and z has different next-hops in the initial and final state. FLIP computes crucial predecessors with a single backward visit (from n to flow sources) of the graph associated to σ.

We also denote the graphs corresponding to the initial and final state for a flow f respectively as G^i_f and G^t_f.

A. Forwarding correctness constraints

A blackhole is defined as the absence of rules for a flow f on a switch b traversed by a forwarding path. Given that the initial and final states are forwarding correct, blackholes can occur during an SDN update if and only if (i) b has no rule for f in either the initial or final state, and (ii) in an intermediate state, a forwarding path for f traverses b while it has no rule for f. Following this observation, for each node b with no rule for a flow f in the state $S_B \in \{init, fin\}$ but with a rule only in $\tilde{S}_B = \{init, fin\} \setminus S_B$, we generate a replacement constraints of the form $\forall p \in cpreds(b, f, \tilde{S}_B)$ $app(rep(b, f)) < app(rep(p, f))$ if $S_B = init$ and $app(rep(b, f)) > app(rep(p, f))$ otherwise. This ensures that (i) if b has no rule before the update ($S_B = init$), it is ready to apply its final rule when any of its final crucial predecessors has installed its final rule, hence whenever a forwarding path can cross b; and (ii) if b has no rule after the update ($S_B = fin$), it keeps its initial rule until all its initial crucial predecessors apply their respective final rules, and a forwarding path cannot cross b anymore. FLIP generates no tag-and-match constraint to avoid blackholes. Indeed, since switches responsible for blackholes do not have rules in the initial or final states, matching operations would coincide with replacement constraints, forcing the application of that single rule throughout the update.

Extracting constraints to avoid evil loops is also quite intuitive. Consider any potential evil loop L for flow f, as obtained by enumerating cycles in the graph $G^i_f \cup G^t_f$. For replacement constraints, we adopt an approach similar to [30]: We identify the set L_{init} of nodes such that their respective next-hops in L are next-hops in the initial but not in the final state. Similarly, the set L_{fin} includes nodes whose next-hop in L is a final but not initial next-hop for the considered flow. In Fig. 1, $v \in L_{init}$ since z is an initial but not final next-hop of v, and $z \in L_{fin}$ for symmetrical reasons. We then generate a replacement constraint forcing any of the nodes in L_{init} to be updated before any of the nodes in L_{fin}. This has already been proved to prevent evil loops during the update [30]. Also, we generalize the intuition used in Fig. 2, and generate tag-and-match constraints imposing that one node in $L_{init} \cup L_{fin}$ matches tags used by its crucial predecessors. Indeed, since both the initial and final states are correct, matching on a single node m in $L_{init} \cup L_{fin}$ provably avoids the evil loop corresponding to L, since m will force packets out of the loop after at most one lap in the loop (as in Fig. 2(b)).

B. Policy preservation constraints

Policy-preservation constraints are the trickiest to identify: No previous work actually provides means to enumerate and formalize them. Abstractly, for every flow f subject to an input policy, FLIP separately colors G^i_f and G^t_f. It then generates constraints based on those colors. In the following, we textually explain how constraints are extracted for any flow f subject to a policy $\mathcal{P}(f)$ and why they are semantically correct. As a reference for explanations, colors assigned by FLIP for cases in Fig. 1 and 4 are reported in Fig. 5 and 6.

First, we color nodes. Given any graph G such that $G = G^i_f$ or $G = G^t_f$, colors are assigned using the following algorithm. First, FLIP identifies all the nodes not having a rule for f in G, and colors them as *blue*. Moreover, by analyzing forwarding paths for f in G, it assigns the *yellow* color to nodes that are not part of any forwarding path (from any source of the flow) even if they have a rule for f. For instance, in the initial graph of Fig. 4, e is blue since it has no rule for f, as shown by Fig. 6; Moreover, v is yellow since it has a rule for f but it is not traversed by any path from a or l (sources of the flow) to d. To determine other colors, FLIP removes from G all the edges part of a satisfied policy path for f (e.g., (v, z) in Fig. 5). Since policies must be satisfied by any path in G, this disconnects G, separating sources and destination into different connected components. FLIP colors all the nodes reachable from any source as *green*, and all the nodes in the connected component of the destination as *white*. Consistently, Fig. 5 shows that FLIP colors s, u and v as green in the initial graph, and z and w as white. By definition, a node g is green if and only if all the paths from g to the destination satisfy $\mathcal{P}(f)$. Symmetrically, a node w is white if and only if all the

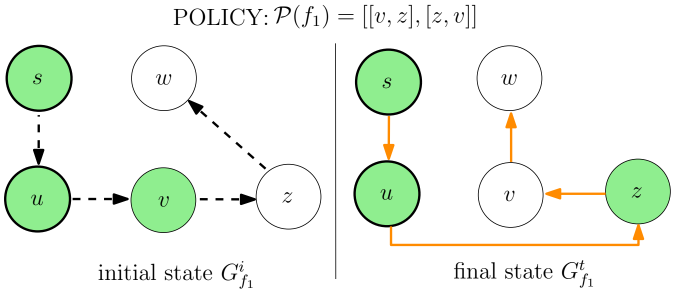

POLICY: $\mathcal{P}(f_1) = [[v, z], [z, v]]$
s
w
u
v
z
initial state $G^i_{f_1}$
final state $G^t_{f_1}$

Fig. 5. Graph coloring performed by FLIP when run on the case in Fig. 1.

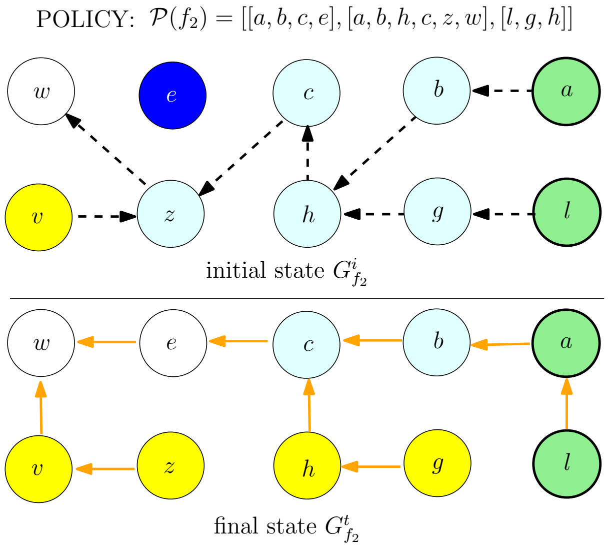

POLICY: $\mathcal{P}(f_2) = [[a, b, c, e], [a, b, h, c, z, w], [l, g, h]]$
w
e
c
b
a
v
z
h
g
l
initial state $G^i_{f_2}$
final state $G^t_{f_2}$

Fig. 6. Graph coloring performed by FLIP when run on the case in Fig. 4.

paths from a source of f to w satisfy $\mathcal{P}(f)$. All the nodes in a connected component that does not include neither sources nor the destination are colored as *cyan*. For example, nodes that are in the middle of a policy path (i.e., excluding the first and the last ones) used to satisfy $\mathcal{P}(f)$ from some sources are cyan. Fig. 6 shows that g, h, b, c and z are cyan in $G^i_{f_2}$ for the example in Fig. 4.

Second, we extract constraints from colored graphs. From node-colored graphs, FLIP extracts several sets of constraints for $\mathcal{P}(f)$, according to Table II. In the table, we use expressions as $n < cpreds(n, f, S)$ instead of $\forall p \in cpreds(n, f, S)$ $app(rep(n, f)) < app(rep(p, f))$ for brevity.

Table II shows that FLIP does not generate constraints for nodes which are either (i) green in both G^i_f and G^t_f, or (ii) white in both G^i_f and G^t_f. Indeed, those nodes cannot be responsible for possible policy violations. Consider a node g which is green in both G^i_f and G^t_f. By definition of green node, $\mathcal{P}(f)$ has to be satisfied by successors of g in both the initial and final state, hence updating g cannot create a violation of $\mathcal{P}(f)$. The same applies to any node w which is white in both G^i_f and G^t_f, since $\mathcal{P}(f)$ has to be satisfied before reaching w in both the initial and final state.

In contrast, constraints are needed for nodes with different colors in G^i_f and G^t_f. Consider, for example, any node r which is white in G^i_f and green in G^t_f, like z in Fig. 5. A rule

	green	**cyan**	**white, yellow**
green	-	n>cpreds(n,f,G^t_f) match on n	n>cpreds(n,f,G^t_f) match on n
cyan	n<cpreds(n,f,G^i_f) match on n	*enum*	n>cpreds(n,f,G^t_f) match on n
white, yellow	n<cpreds(n,f,G^i_f) match on n	n<cpreds(n,f,G^i_f) match on n	-

n=analyzed node, f=flow, G^i_f=initial state, G^t_f=final state

TABLE II
FLIP CONSTRAINT EXTRACTION FOR ANY NODE n, WITH INITIAL AND FINAL COLORS SPECIFIED BY ROWS AND COLUMNS, RESPECTIVELY. NO CONSTRAINT IS GENERATED IF n IS BLUE IN THE INITIAL OR FINAL STATE.

replacement on r can induce a policy violation from a given source in G^i_f: Indeed, the initial policy path can be bypassed via the final path from the source to r (e.g., $[s, u, z]$), and the final policy path can be circumvented with the initial path from r to the destination (e.g., $[z, w]$). FLIP constrains the rule replacement on r to be applied before replacements on any of its initial crucial predecessors. This guarantees that no source can reach r with a final path before r uses its final rule; in turn, this prevents policy paths to be bypassed, as in the example above. In Fig. 5, FLIP indeed adds a replacement constraint $app(rep(z, f_1)) < app(rep(u, f_1))$. If respected, this constraint ensures that during the update either (i) u uses its initial rule, and the initial, policy-compliant path is followed from u to z; or (ii) u uses its final rule and z uses its own final rule as well, hence the policy is satisfied after z (since it is green in the final state). With a similar rationale, we generate a tag-and-match constraints in which r matches tags added by its initial and final crucial predecessors.

Similar arguments prove the need for constraints for nodes with other combinations of (different) colors in G^i_f and G^t_f.

FLIP only makes one exception, for nodes that are cyan in both G^i_f and G^t_f (like b and c in Fig. 6) since they have to be treated differently. For those nodes, even computing whether constraints are needed is not obvious, because their presence in paths violating of $\mathcal{P}(f)$ depends on possible next-hops of both their respective predecessors and successors. Hence, FLIP enumerates all paths in $G^i_f \cup G^t_f$ that contain at least one node which is cyan in both states. This is a sort of limited path enumeration, which is restricted on the basis of potentially-dangerous nodes (cyan in both states) belonging to complex policy paths (with more than two nodes). This enables FLIP to pinpoint the paths among the enumerated ones that violate $\mathcal{P}(f)$. This way, FLIP detects that $[a, b, c, z, w, d]$ is a possible forwarding path for f_2 which violates $\mathcal{P}(f_2)$ in Fig. 4. Once a policy-violating path V is found, FLIP generates a replacement constraint on a specific node s, such that the sub-path of V ending in a next-hop of s is not included in any policy path for the considered flow. In Fig. 4, c is the constrained switch for $V = [a, b, c, z, w, d]$, since $[a, b, c, z]$ is not included in any policy path in $\mathcal{P}(f_2)$. In particular, FLIP constrains c's rule replacement to be applied before its crucial predecessor on V, i.e., b in this case. With a similar rationale, FLIP also adds a tag-and-match constraint in which the same switch used for the replacement constraint (c in our example) matches and all its crucial predecessors in both G^i_f and G^t_f tag.

C. Tracking relationships between constraints

FLIP also identifies alternative and dependent constraints.

FLIP stores constraints generated by the same potential violation as alternative. This generalizes the intuition used in §II to produce the operational sequence shown in Fig. 2. In the generation of that sequence, a key observation is that the evil loop between v and z can be broken by either (i) replacing v's rule before replacing z's one, (ii) tagging on v and matching on z, or (iii) tagging on z and matching on v. Consistently, FLIP records those constraints as alternative. More generally, FLIP stores as alternative all the set of constraints generated for the same blackhole, evil loop or policy violation.

FLIP tracks dependencies between constraints. Such dependencies are needed to guarantee that a tag τ is not overwritten or removed before reaching the node which has to match τ – an implicit assumption behind tag-and-match constraints.

To avoid harmful tag overwriting, FLIP creates a dependency between the original tag-and-match constraint and a tag-and-match constraint involving the node that can incorrectly modify the tag. More precisely, whenever a tag may traverse a node n that can overwrite it before reaching all its corresponding matching nodes, FLIP introduces a constraint dependency to impose that n matches and preserves the tag.

Tags can be potentially overwritten en route in two cases.

The first case is represented by nodes that are critical to prevent both a loop L and a violation of a policy P. For example, consider again Fig. 1, and assume that we need to preserve strong consistency, i.e., ensure that either the initial path $[s, u, v, z, w]$ or the final one $[s, u, z, v, w]$ is followed. A tag-and-match constraint in which v tags and z matches avoids the evil loop between v and z. However, the tag set by v (to exit the evil loop) may overwrite the one set by u (to enforce strong consistency). This is exactly what happens in Fig. 2(b): In that case, however, the tag set by u was intended to enforce the policy subpath $[z, v]$, hence to be propagated up to v. In contrast, overwriting the tag at v would disrupt strong consistency, as the latter requires that all nodes consistently match the tag set by u until the destination is reached. Hence, FLIP stores the tag-and-match constraint with v matching as dependent on the tag-and-match constraint where z matches. More complex scenarios involving nodes cyan in both the initial and final states are identified during the enum procedure in the extraction of policy constraints (see Table II).

The second case where dependencies are needed is represented by nodes participating in nested evil loops. Consider Fig. 7. There are two nested evil loops here: $[a, b, c, e, a]$ and $[a, b, e, a]$. When FLIP extracts the tag-and-match constraint where e matches, it also detects that the tags may be overwritten. Indeed, both b and c are crucial predecessors of e, hence they are selected as taggers. However, because of their relative position in the sub-path $[a, b, c, e]$, c could override a tag set by b. To avoid such an overwriting, FLIP introduces constraint dependencies that force tags to be propagated throughout any loop. In the example of Fig. 7, FLIP therefore creates a dependency between the tag-and-match constraint where e matches and those where b and c match.

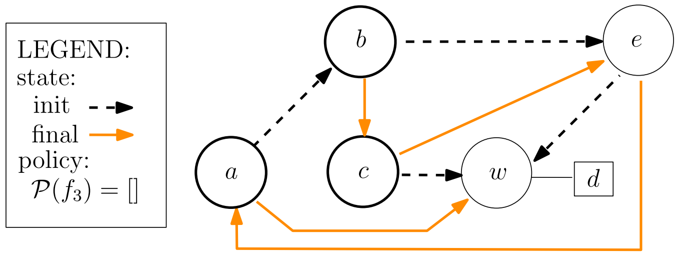


Fig. 7. Example of a topology where dependencies are needed.

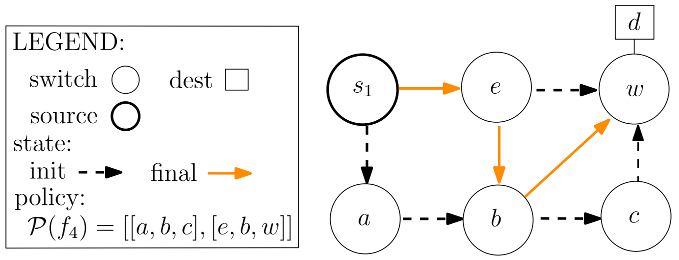


Fig. 8. Example where a constraint swap requires rewriting of a replacement constraint.

V. FLIP Constraint Swapping

Starting from a set of active constraints, this procedure swaps an active constraint with one of its alternatives. Selecting the constraints to swap can be done in different ways. FLIP uses a heuristic approach that efficiently finds a safe sequence with few matching operations, to limit the memory overhead. This heuristic does not guarantee to find an update sequence with the minimum number of extra rules – which remains an open research problem. Nevertheless, it shows very good performance in practice (see §VI). Details follow on how FLIP constraint swapping ensures efficiency and correctness.

FLIP always swaps replacement constraints with tag-and-match ones, never the opposite. This means that replacement constraints are never added back, i.e., swapping a replacement constraint translates into permanently discarding it. This strategy is guaranteed to eventually converge because all matching constraints are set as active in an extreme case. Also, it implies that FLIP falls back to the two-phase commit approach [28] in the worst case.

FLIP selects constraints to be swapped so that it quickly finds a solvable set of constraints. Indeed, at each invocation of the constraint swapping procedure, FLIP selects a pair of constraints (R, M), where R is the replacement constraint to be swapped with the M tag-and-match one, in such a way that (i) R is in an Irreducible Infeasible Set [5] of the active constraints, i.e., a minimal set of active constraints that cannot be satisfied simultaneously; and (ii) M has the minimal number of dependent constraints among the alternatives for R.

FLIP preserves the semantics of all constraints after any swap. After having selected the pair of constraints (R, M) to be swapped, FLIP updates all active constraints to take into account the effect of the swap. This involves multiple actions.

First, any replacement constraint R' with M as an alternative is removed from the active constraints. Indeed, the potential anomalies that R' avoids are now prevented by M.

Second, FLIP adds all M's dependencies to the set of active constraints, i.e., respecting the meaning itself of such dependencies. This also implies removing other replacement constraints having one of M's dependencies as alternatives. Note that the need for setting dependent constraints as active intuitively justify our selection heuristic that selects the alternative to R with a minimal number of dependencies.

Third, FLIP rewrites replacement constraints (if any) involving the switch r which matches in M. More precisely, each replacement constraint $app(rep(x, f)) < app(rep(r, f))$, where x is a switch different from r, is replaced by a set of constraints $app(rep(x, f)) < app(rep(y, f))$, one for every crucial predecessor y of r. This is needed to preserve the semantics of the original constraint, that is, to prevent r from using its final rule if x was still using its initial one. Since r uses either its initial or final rule depending on packet tags, the new constraints indeed impose that the final rule is installed on x before y installs its final rule (and adds final tags), therefore indirectly forcing r to use its final rule too. We apply a similar rewriting for constraints $app(rep(x, f)) > app(rep(r, f))$.

Fig. 8 exemplifies a constraint rewriting performed by FLIP and illustrates why it is necessary. The input policy is meant to enforce strong consistency (only the initial or final paths are acceptable). For s_1 to use its final path upon its rule replacement, we would need constraints $app(rep(b, f_4)) < app(rep(e, f_4))$ and $app(rep(e, f_4)) < app(rep(s_1, f_4))$ to be satisfied. Also, for s_1 to keep using the initial path before its rule is replaced, $app(rep(s_1, f_4)) < app(rep(b, f_4))$ must hold. This implies that a safe update cannot carried out with rule replacement only. Thus, FLIP swaps one of those replacement constraints with an alternative tag-and-match one. Assume that FLIP selects $app(rep(e, f_4)) < app(rep(s_1, f_4))$ as the constraint to be swapped: Its alternative is to tag on s_1 and match on e. Now, the rule applied by e depends on the tag set by s_1. Hence, FLIP rewrites $app(rep(b, f_4)) < app(rep(e, f_4))$ as $app(rep(b, f_4)) < app(rep(s_1, f_4))$. Such a rewriting is fundamental to maintain correctness, i.e., to keep the property that s_1 uses its final path upon its rule is replaced. Indeed, without rule rewriting, we would have left with replacement constraints $app(rep(b, f_4)) < app(rep(e, f_4))$ and $app(rep(s_1, f_4)) < app(rep(b, f_4))$: According to those constraints, we could have started with a rule replacement at s_1, which however would have installed path $[s_1, e, b, c, w, d]$ (since e matches the final tag set by s_1) and violated the input policy $\mathcal{P}_4$. In contrast, the rule rewriting leaves FLIP with constraints $app(rep(b, f_4)) < app(rep(s_1, f_4))$ and $app(rep(s_1, f_4)) < app(rep(b, f_4))$, which are still unsatisfiable and for which we need another constraint swap. Eventually, FLIP finds a safe sequence based on matching on both e and b and tagging on s_1.

A complete illustration of how constraint swapping works for the case in Fig. 1 is reported in Fig. 9. This constraint swap leads to the solution displayed in Fig. 2. In the figure, the first set of constraints (top left of the figure) is the one extracted by FLIP from the original update problem. Initially,

constraints

cause	active constraints	alternatives
loop (v, z)	$app(rep(v, f_1)) < app(rep(z, f_1))$	*match on z* *match on v*
policy $\mathcal{P}$	$app(rep(u, f_1)) < app(rep(v, f_1))$	*match on v*
	$app(rep(z, f_1)) < app(rep(u, f_1))$	*match on z*

LP

$$\begin{array}{l} \min\ r_u + r_v + r_z + r_w \\ r_v < r_z \\ r_u < r_v \\ r_z < r_u \\ r_u, r_v, r_z, r_w > 0 \end{array}$$

$\downarrow$ **swap** $app(rep(v, f_1)) < app(rep(z, f_1))$ with *match on z*

cause	active constraints	alternatives
loop (v, z)	*match on z*	
policy $\mathcal{P}$	$app(rep(u, f_1)) < app(rep(v, f_1))$	*match on v*

$$\begin{array}{l} \min\ r_u + r_v + r_z + r_w \\ r_u < r_v \\ r_z > r_u, r_v, r_w \\ r_u, r_v, r_z, r_w > 0 \end{array}$$

Fig. 9. A constraint swapping solving the scenario in Fig. 1.

all and only replacement constraints are active. FLIP translates active constraints into the linear program (LP) shown at the top right of Fig. 9, where r_x stands for $app(rep(x, f_1))$, with $x \in \{u, v, z\}$. Such an LP has no solutions. The swapping procedure selects $app(rep(v, f_1)) < app(rep(z, f_1))$ as constraint to be swapped, since it is in the set of contradictory constraints. Hence, it updates the set of active constraints by removing the constraint to be swapped and adding one of its alternatives, namely *match on z*. Further, $app(rep(z, f_1)) < app(rep(u, f_1))$ is also removed from the active constraints, since *match on z* was an alternative to it. No other constraint is added or modified because *match on z* does not have dependencies and z is not involved in any other replacement constraint. The LP deriving from the new set of active constraints is shown in the bottom right part of the figure. In this LP, $r_z > r_u, r_v, r_w$ derives from the formalization of the match-and-tag constraints on z, as discussed in §III-C. Note that *match on z* also implies other constraints, imposing that $tag(v, f_1, \tau)$ and $match(z, f_1, \tau, \emptyset)$ must be in the first two update steps. Since they do not impose constraints on any other operation, FLIP does not include the latter constraints into the LP but accommodate them by post-processing the LP solution, i.e., adding those operations to the very first steps of the returned sequence.

VI. Evaluation

We evaluate FLIP by performing $50,000$ experiments. In each experiment, we generate an update problem; on each problem, we run our FLIP implementation, which is available at *http://inl.info.ucl.ac.be/softwares/flip*. We verify that the operational sequence computed by FLIP is correct by simulating its application to the corresponding network. To this end, we apply one operation at the time, following the sequence generated by FLIP, and we check forwarding correctness and policy preservation after each operation. For efficiency reason, we apply operations in the same step in a random order rather than simulating all possible permutations. While this can theoretically lead to false positives (i.e., sequences accidentally considered correct), the sheer number of experiments provides statistical confidence on the absence of false positives. We focus on single-flow updates, since FLIP works on a per-flow basis (see Fig. 3).

A. Setup

As dataset, we use all the publicly available Rocketfuel topologies [29], denoted by their identifiers (1221, 1239, 1755, 3257, 3967, 6461) in the following. Their sizes range from 79 nodes and 294 edges to more than 300 nodes and almost 2,000 edges. For each topology, we select uniformly at random a node as destination, and a random 10% of the nodes as sources. All the equal-cost (ECMP) shortest paths from any source to the destination in the original topology are taken as the initial state. Further, we randomly pick 80% of the links and set their weight to a value chosen uniformly at random among the weights of the original topology (possibly its initial one). The ECMP shortest paths from the sources to the destination in this reweighted graph are taken as the final state.

This methodology provides qualitatively diverse update scenarios. Depending on the selected links and new weights, reweighting links at random statistically tends to generate (i) cases where only a few paths change, (ii) major routing modifications where most nodes change next-hop, and (iii) intermediate scenarios between those two extremes. Table III shows that this is indeed the case in our experiments: In half of them, the nodes changing next-hop are roughly between 60% (first quartile) and 45% (third quartile), although only 20% of them need to be updated in 5% of the experiments.

experiments (percentile)	**5th**	**25th**	**50th**	**75th**	**95th**
next-hop changes	66.67%	59.42%	50.63%	45.34%	20.19%

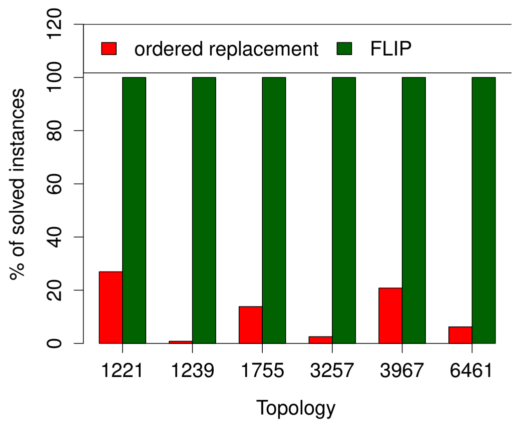
TABLE III
Percentage of nodes changing next-hops in our experiments

Finally, we add random policies so that every path from a source to the destination complies with at least one policy. We choose non-trivial policies composed of paths longer than 2 nodes, which also shows FLIP's support for more complex policies than single-node traversal ones considered by [18].

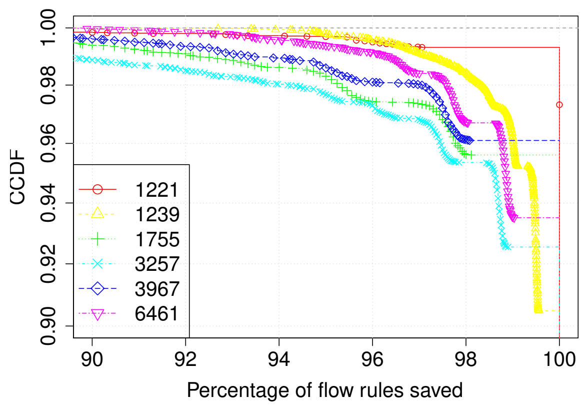
B. Results

The results of our experiments are summarized in Fig. 10-11. We now discuss those results in more details.

FLIP always computes safe updates. It prevents any possible blackhole, evil loop or policy violation in each and every experiment. FLIP's 100% success rate marks an important difference with previous ordered replacement techniques, like [20], that preserve policies by ensuring strong consistency, i.e., using either the initial or the final paths for each flow. We run an exhaustive search approach to compute the number of cases in which strong consistency can be guaranteed by ordered rule replacements. Results are displayed in Fig. 10(a). They show that ordered replacement techniques cannot find an operational sequence in more than $\approx 25\%$ of our experiments. Even worse, their success rate greatly depends on the specific topology, and larger topologies (e.g., 1239) are virtually impossible to tackle. In contrast, FLIP finds a safe sequence in all our update scenarios. This is because FLIP explores a much larger solution space, including operational sequences tailored to



(a) Success rate of FLIP and of ordered replacement techniques like [20].



(b) Additional rules that FLIP saves relatively to two-phase commit techniques like [10], [12], [28].

Fig. 10. FLIP outperforms previous approaches in our 50,000 experiments on Rocketfuel topologies.

guarantee the input policy (rather than strong consistency) and combining rule replacements with tagging-and-matching operations (rather than restricting to the former ones).

FLIP hugely reduces the number of added rules. In the 99.9th percentile of the experiments, FLIP adds one rule to 8.7% of the nodes. We now compare FLIP's overhead with the one of two-phase commit techniques. For each experiment, we compute the number N_{DUP} of additional rules added by [28], and the number N_{FLIP} of additional rules added by FLIP. To be fair, we assume that the two-phase commit approach does not match on nodes with the same next-hops in the initial and final states, as also suggested in [28]. We then calculate the percentage of rules saved by FLIP as $\frac{N_{DUP}-N_{FLIP}}{N_{DUP}} \times 100$. This percentage expresses the relative comparison of the overhead induced by FLIP and two-phase commit, with a metric normalized with respect to topology sizes.

Fig. 10(b) shows the Cumulative CDF of such percentage in our experiments. A data point (x, y) in the plot indicates that for a fraction y of the experiments, FLIP saves at least x% of the rules that would be used by [28]. Across all topologies, *in 98% of the experiments ($y = 0.98$) FLIP saves at least 94% ($x = 94$) of the rules added by [28].* Across all our experiments, at least 87.8% of the rules are saved by FLIP.

Note that FLIP's savings are fundamentally different from those of previous variants of two-phase commit techniques. Prominently, [12] proposes to reduce the update overhead by updating groups of flows in different rounds. In contrast to FLIP, this workaround *does not avoid rule additions*, but only distributes them over time (e.g., see Table I and the corresponding discussion). Moreover, [12] degenerates to [28] in our experiments, since a single flow is updated in them.

FLIP computes fast updates. Fig. 11 shows a CDF of the number of update steps in our experiments. In all our experiments, the median number of update steps is 5, the 95th percentile is 8, and the 99.9th percentile 12. This distribution does not vary excessively across the different topologies. The only exception is represented by 1221, the smallest topology, where FLIP's sequences have less than 4 steps in 95% of the experiments. As a comparison, the most generally applicable

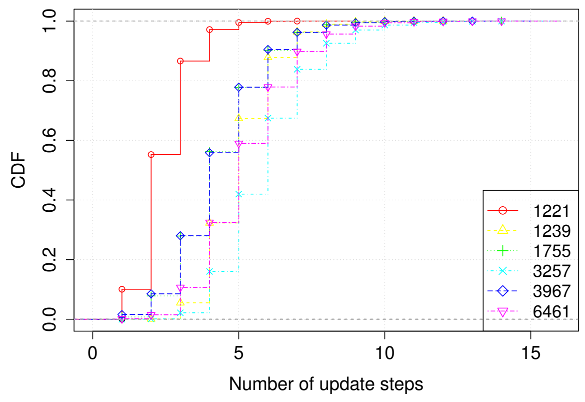


Fig. 11. FLIP sequences are composed of a few steps, a dozen at most.

two-phase commit technique [12] computes updates with 3 steps per round, where each round updates a subset of the flows (see Table I and its discussion). The number of rounds (hence, of steps) is determined by the additional memory available at the most constrained switch involved in the update. If 1,000 flows have to be updated and there is at least one switch involved in the update which cannot accept 1,000 additional entries, then at least 2 rounds are needed, i.e., 6 steps (already more than FLIP's median). If there is at least one switch which cannot accept 500 entries, then at least three rounds and 9 steps are needed, and so on.

The small number of update steps in FLIP sequences is due to the very design of our algorithm. When computing rule-replacement orderings, FLIP minimizes the sum of the steps to which rule replacements are assigned (e.g., see the objective function of the LPs in Fig. 9). This also implies that FLIP sequences have the same number of steps with respect to an optimal sequence computed by ordered replacement techniques, when such a sequence exists. Fig. 10(a) indicates that this is the case in about 10% of our experiments.

FLIP often terminates in sub-seconds. Fig. 12 shows a CDF of FLIP running times in our experiments. FLIP's median execution time is 0.176 seconds when run on a commodity

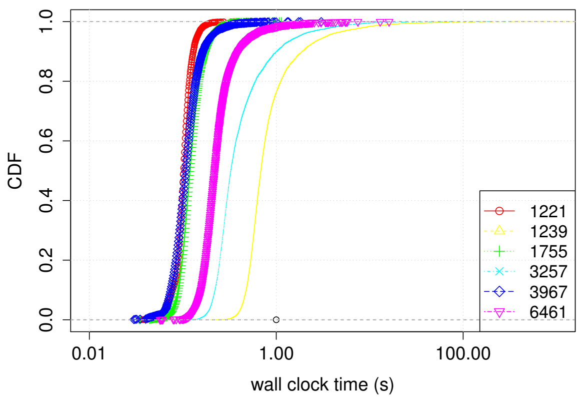


Fig. 12. FLIP execution time is often sub-second.

server (8-core 2.66GHz CPU[2] and 16 GB of RAM). Also, 94% of the instances are solved in less than 1 second, and 99% in less than 4 seconds. The topology with the worst performance is 1239, the largest one, where the 95th percentile of the execution time is 3.38 seconds and the 99th is 15 seconds.

Those results show that FLIP readily supports many update scenarios, from deployment of policy changes to online traffic engineering (typically performed at the timescale of few minutes [8]) to pre-computation of failure reaction. Code optimization and more powerful hardware likely improve FLIP runtime, and make the algorithm suitable for bigger networks.

Another important observation is that FLIP execution time does not dominate the overall time of an update. In fact, the total time to complete an update is the sum of computing the operational sequence (FLIP execution time, in our case) and applying such operations in the network. The latter is equal to the number of update steps (see Fig. 11 for FLIP) multiplied by the time to apply the operations in each step, which in turn depends on factors such as network latency, message processing time at the switches, rule installation time at the switches, reception of acknowledgements at the controller, and possibly error recovery and retransmissions (e.g., for lost messages). Since those factors are network-specific, it is hard to give a general estimation of the sequence application time; however, recent studies (e.g., [10], [35]) suggest that it is realistically in the order of seconds per step (irrespective of the update technique used to compute the migration plan), that is, much more than the FLIP execution time.

VII. Discussion

We now discuss FLIP limitations, and possibilities to mitigate them through variants of the current algorithm.

Scope. While FLIP is a general algorithm that can be used in any network setting and update scenario, it may be an over-sophisticated solution in some cases.

First, FLIP is not necessarily advantageous for specific subsets of policies. For instance, more specialized and efficient algorithms [4], [33] have been recently proposed for the special case of updates solely admitting initial and final paths for every flow throughout the update (strong consistency).

[2]Our FLIP implementation is single-threaded, but the used LP solver libraries rely on parallel code

Second, FLIP provides the most benefits when initial and final paths for the same flow can differ arbitrarily, as for generic updates of enterprise, wide area (WAN) or service provider networks (see §VI). In contrast, initial and final paths for the same flow tend not to form cycles in data center (DC) networks, because of the structural regularity of typical DC topologies (like fat trees [2]). For example, the case depicted in Fig. 1 is topologically impossible in a fat tree topology. This tends to make replacement-only techniques always applicable, and FLIP's main features (like modeling and swapping of constraints on rule replacements and additions) unnecessary.

We double-checked the limited benefits of our approach for DC updates by running FLIP on 100 update scenarios on synthetic fat tree topologies. In our experiments, we significantly reshuffled paths assigned to traffic flows by reweighting 80% of the links (randomly extracted), i.e., multiplying the weight of each of those links by a value randomly extracted among 2, 5, 10, or 25. In all those experiments, FLIP always returned a sequence with only rule replacements.

Flexibility on update constraints and objectives. A multitude of constraints and objectives may be desirable for real-world updates. FLIP currently focuses on minimization of the update sequence, subject to its guaranteed safety.

Nevertheless, by internally relying on LPs, FLIP readily supports several customizations of update constraints and objectives. For example, different weights can be assigned to LP constraints in order to privilege early update of certain switches over others, e.g., depending on their operational importance (volumes of carried traffic) or reactivity [10].

FLIP can also be extended to provide additional flexibility. By tweaking constraint extraction and swapping procedures, it can be customized to support higher-level preferences on the returned update sequences. For example, we can forbid rule additions on specific nodes by never generating matching constraints on those nodes. Similarly, we can avoid adding rules to prevent specific policy violations (e.g., enforcing low latency paths) by never swapping ordering constraints for such policies with any matching constraint.

Time complexity. FLIP includes sub-functions whose complexity is not polynomial with respect to the size of the input. In particular, the constraint extraction procedure sometimes requires to enumerate paths between sets of nodes (see Table II). Also, in the swapping phase, FLIP computes minimal sets of LP constraints that cannot be satisfied together, which is a computationally hard problem [5].

Our evaluation on Rocketfuel topologies (§VI) shows that worst-case time complexity tends not to be a problem for relatively sparse topologies like WANs (see Fig. 12). Nevertheless, potential time inefficiency may become more critical for real-time updates (e.g., reaction to failures). Also, it can lead to limited practicality in very dense topologies, like data center ones (which however are not the settings where FLIP is mostly useful, see above): While the median execution time has been $91ms$ in our DC experiments, FLIP constraint extraction took several minutes in a few cases, because nested loops induced many dependencies between loop constraints.

Whenever time efficiency is critical, slight algorithmic variations (e.g., based on domain knowledge) can improve FLIP performance. For instance, we can cut down the time needed to deal with loop-constraint dependencies by discovering those dependencies at runtime (when a constraint swap is needed), rather than beforehand. This works especially well for data-center networks, where constraints are rarely or never swapped (see above). Additionally, we can trade optimality and flexibility for shorter execution time. For example, we could internally replace LPs with dependency graphs, having one node per rule replacement and one edge per constraint between replacements. We would then extract a sequence by using a topological sorting algorithm instead of an LP solver. The dependency graph model would enable us to replace the expensive IIS procedure with a polynomial-time visit on the graph where we check for a single loop. This performance gain would however come at the cost of (i) potential sub-optimality of the solution returned by the topological sorting algorithm, e.g., in terms of sequence length; (ii) less flexibility with respect to custom constraints and objectives (see previous discussion); and (iii) impossibility to guarantee that the set of infeasible constraints is minimal, which may eventually lead to swap more constraints. A full investigation of those variations is left for future work.

VIII. Conclusions

In this paper, we studied how to achieve safe, efficient and robust updates of SDN networks, that preserve both forwarding correctness and input policies. We unveiled the power of combining rule replacements and additions, demonstrating how such combinations can overcome the limitations of existing approaches. Also, we showed how to systematically exploit this power. We presented FLIP, an algorithm that interleaves rule replacements and additions to create safe operational sequences. Our extensive evaluation shows the entity of the gain achieved by FLIP with respect to previous approaches. In our WAN experiments, FLIP is 90% more efficient than two-phase commit techniques in terms of memory overhead, and it supports 90% more update scenarios than ordered replacement ones. Our experiments also show that FLIP quickly computes updates terminating in a very limited number of steps.

The model that FLIP uses to reason about combinations of rule replacement and additions makes FLIP extensible. For instance, FLIP can easily support domain-specific constraints such as memory restrictions on specific switches. We successfully tested one of such cases, in which we prevented any rule addition on a specific switch by manually injecting an additional constraint to FLIP's model.

References

[1] I. F. Akyildiz et al. A Roadmap for Traffic Engineering in SDN-OpenFlow Networks. *Computer Network*, 71:1–30, 2014.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[3] S. Brandt, K.-T. Forster, and R. Wattenhofer. On Consistent Migration of Flows in SDNs. In *Proc. INFOCOM*, 2016.

[4] P. Černý, N. Foster, N. Jagnik, and J. McClurg. Optimal Consistent Network Updates in Polynomial Time. In *Proc. DISC*, 2016.

[5] J. W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Springer, 2007.

[6] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of Consistent Network Updates. arXiv:1609.02305, 2016.

[7] S. Ghorbani and M. Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *Proc. HotSDN*, 2012.

[8] R. Hartert et al. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *Proc. SIGCOMM*, 2015.

[9] C.-Y. Hong et al. Achieving High Utilization with Software-driven WAN. In *Proc. SIGCOMM*, 2013.

[10] X. Jin et al. Dynamic Scheduling of Network Updates. In *Proc. SIGCOMM*, 2014.

[11] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient Traffic Splitting on Commodity Switches. In *Proc. CoNEXT*, 2015.

[12] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[13] M. Kuzniar, P. Peresini, and D. Kostić. Providing Reliable FIB Update Acknowledgments in SDN. In *Proc. CoNEXT*, 2014.

[14] A. Liu, C. Meiners, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, April 2010.

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

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

[17] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling Loop-free Network Updates: It's Good to Relax! In *Proc. PODC*, 2015.

[18] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *Proc. HotNets*, 2014.

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

[20] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient Synthesis of Network Updates. In *Proc. PLDI*, 2015.

[21] R. McGeer. A Safe, Efficient Update Protocol for Openflow Networks. In *Proc. HotSDN*, 2012.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[23] T. Mizrahi and Y. Moses. Software defined networks: Its about time. *Proc. INFOCOM*, 2016.

[24] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Scheduling Network Updates with Timestamp-based TCAM Ranges. In *Proc. INFOCOM*, 2015.

[25] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *Proc. NSDI*, 2016.

[26] Z. A. Qazi et al. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. SIGCOMM*, 2013.

[27] M. Reitblatt et al. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *Proc. HotSDN*, 2013.

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[29] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. SIGCOMM*, 2002.

[30] L. Vanbever et al. Seamless Network-Wide IGP Migrations. In *Proc. SIGCOMM*, 2011.

[31] S. Vissicchio and L. Cittadini. Flip the (flow) table: Fast lightweight policy-preserving SDN updates. In *Proc. INFOCOM*, 2016.

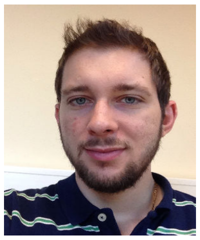
[32] S. Vissicchio et al. On the Co-Existence of Distributed and Centralized Routing Control-Planes. In *Proc. INFOCOM*, 2015.

[33] S. Vissicchio et al. Safe Update of Hybrid SDN Networks. *IEEE/ACM Transactions on Networking*, 25(3):1649–1662, june 2017.

[34] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu. Compiling Minimum Incremental Update for Modular SDN Languages. In *Proc. HotSDN*, 2014.

[35] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu. RuleTris: Minimizing Rule Update Latency for TCAM-Based SDN Switches. In *Proc. ICDCS*, 2016.

[36] J. Yu. Scalable Routing Design Principles. RFC 2791, 2000.



Stefano Vissicchio is a Lecturer at University College London. He obtained his Master degree from the Roma Tre University in 2008, and his Ph.D. degree in computer science from the same university in April 2012. Before joining University College London, he has been postdoctoral researcher at the Université catholique of Louvain. Stefano's research interests span network management, routing theory and protocols, measurements, and new network architectures like Software Defined Networking. He has received several awards including the ACM SIGCOMM 2015 best paper award, the ICNP 2013 best paper award, and two IETF/IRTF Applied Networking Research Prizes.

Luca Cittadini received his master degree from the Roma Tre University in 2006, and a Ph.D. degree in Computer Science and Automation from the same institution in 2010, defending the Thesis "Understanding and Detecting BGP Instabilities". During his Ph.D. he was a teaching assistant in the computer network research lab. His research activity is primarily focused on routing: Luca has worked on both intra-domain and inter-domain routing topics, including theoretical analysis of the BGP protocol, configuration and reconfiguration techniques, as well as active and passive measurements.