FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates

Stefano Vissicchio
Université catholique de Louvain
ste.vissicchio@gmail.com

Luca Cittadini
RomaTre University
luca.cittadini@gmail.com

***Abstract*—We propose FLIP, a new algorithm for SDN network updates that preserve forwarding policies. FLIP builds upon the dualism between replacements and additions of switch flow-table rules. It identifies constraints on rule replacements and additions that independently prevent policy violations from occurring during the update. Moreover, it keeps track of alternative constraints, avoiding the same policy violation. Then, it progressively explores the solution space by swapping constraints with their alternatives, until it reaches a satisfiable set of constraints. Extensive simulations show that FLIP outperforms previous proposals. It achieves a much higher success rate than algorithms based on rule replacements only, and massively reduces the memory overhead with respect to techniques solely relying on rule additions.**

I. Introduction and Related Work

Consider an SDN network where the forwarding has to be updated, as can often happen, e.g., to better balance traffic load, steer some flows through virtualized functions or accommodate new security policies. The forwarding is determined by per-flow rules that SDN switches apply to packets. In the update, the controller has to instruct switches to add, change or remove some rules. To avoid service disruption, both forwarding correctness (i.e., packet delivery) and policies (i.e., requirements on forwarding paths) have to be preserved throughout the update. Moreover, the update strategy must be robust with respect to factors (unpredictable to the controller) like non-deterministic processing time on the switch to install or modify rules, or delayed messages between the controller and the switches. This excludes naive strategies like pushing the final rules to the switches at the same time. Rather, the controller has to apply a carefully-computed sequence of operations, so that any single operation can be independently rolled forward or back with no impact on policies.

Despite the abundance of literature on this topic, none of the existing techniques supports policy-preserving updates *efficiently*. Some proposals focus on congestion avoidance [5], [3], [10] or forwarding correctness [12], [22], and do not support policy preservation at all. Among the previous contributions that do preserve policies, *ordered replacement* techniques [13], [11] compute a specific order to replace rules. They are efficient but their applicability is limited: It is known that an order that guarantees both forwarding and policy preservation might not exist [11]. Another approach [18], [7], [6] consists in installing both the initial and final rules on all switches, and tagging packets to signal which rules should be applied. We refer to this approach as *two-phase commit*. While this natively preserves both correctness and policies, it is highly inefficient, to the point to be unpractical [7], [14]. Its main drawback is that it doubles the number of rules on every switch, wasting precious TCAM memory which is a scarce, expensive, and power-hungry resource [9]. Switch memory may be rather needed to deal with the always growing number of services or to guarantee good network performance, e.g., implementing (i) fine-grained traffic engineering, (ii) fast reaction to security attacks, or (iii) fast failure recovery [17].

In this paper, we study how to compute operational sequences that preserve forwarding correctness and policies, using additional rules only when necessary. We unveil the degrees of freedom opened by the inter-changeability between rule replacements and additions in preventing a correctness or policy violation. Moreover, 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 meet correctness and policy requirements, e.g., by temporarily admitting forwarding paths with loops that are traversed only once by packets before they are correctly delivered to the destination.

Unsurprisingly, this additional expressiveness comes at the cost of making the safe update problem more challenging. First, it significantly increases the search space, e.g., because a much higher number of solutions are possible (all combinations between rule replacements and additions). Second, finding a safe sequence implies understanding the interactions between rule replacements and additions applied to different switches (e.g., distinguishing loops that are crossed only a finite number of times from those disrupting connectivity).

We address those challenges with an original algorithm, FLIP. To compactly represent the search space and quickly compute an operational sequence, FLIP formalizes possibilities to avoid correctness violations as constraints on rule replacements and additions. Moreover, it discovers relationships between those constraints. Notably, it identifies alternative constraints. For example, given a potential policy violation, FLIP can determine that either constraints A and B need to 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.

Stefano Vissicchio is a postdoctoral researcher of the Belgian fund for scientific research (F.R.S.-FNRS)

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

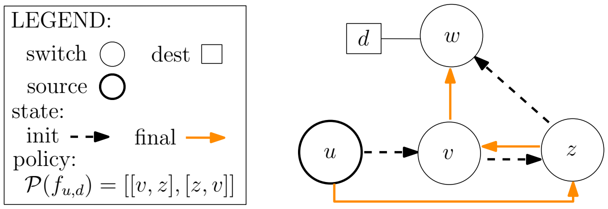


Fig. 1. An update scenario with a policy to be preserved.

The rest of the paper details the following contributions.

Analysis (§II). We detail how combining rule replacements and additions opens additional degrees of freedom in the policy-preserving update problem. Also, we show how they 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 on operations and their relationships.

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

Experimental evaluation (§VI). We evaluate our implementation of FLIP by simulating $50,000$ update scenarios for realistic networks. Our results show that FLIP hugely 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 for brevity) has to update the controlled network. For the sake of the example, the controller has to modify the forwarding only for the flow $f_{u,d}$ of packets sourced at u 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 to switches. Specifically, it can add, modify or delete flow rules used by a switch to process packets belonging to $f_{u,d}$. We distinguish three types of operations. A *rule replacement* operation $rep(s, f)$ instructs a switch s to replace its current rules for 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 θ, and represents the absence of a tag. Both rule replacement and tagging operations modify an existing rule, hence the number of installed rules does not change after the operation is applied. Conversely, a matching operation involves adding a new rule, and consumes an additional slot in the TCAM memory of the affected 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 guaranteed to be delivered to d; and (ii) policies are satisfied throughout the update. In our case, the policy $\mathcal{P}(f_{u,d})$ (see left side of Fig. 1) imposes that packets belonging to $f_{u,d}$ must traverse link (v, z) in either of the two directions.

Despite both the initial and final states guarantee packet delivery and satisfy $\mathcal{P}(f_{u,d})$, those properties can be violated during the update, depending on the order in which operations are applied to switches. In Fig. 1, for example, if the first operation is replacing the rule on z, $rep(z, f_{u,d})$, then packets for flow $f_{u,d}$ are trapped in a permanent loop between v (that applies its initial rule) and z (that applies its final rule) after $app(rep(z, f_{u,d}))$. The loop persists until $app(rep(v, f_{u,d}))$. Similarly, if $rep(u, f_{u,d})$ is the first operation, then $f_{u,d}$ is forwarded over path $[u, z, w, d]$, hence violating the policy.

A. Previous techniques have limitations

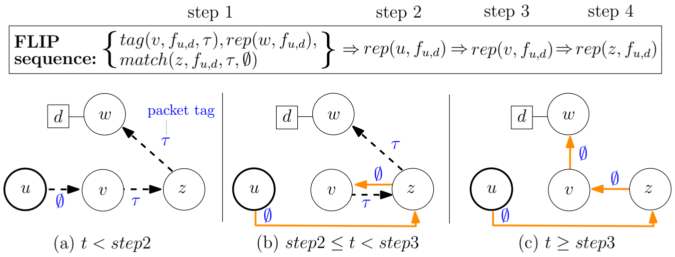
Prior work achieves safe updates by either (i) computing a proper sequence of rule replacements, when it exists (e.g., [13], [11]); or (ii) applying matching operations on all switches and progressively applying tagging operations on flow entry points (u in our example) in order to make all the switches use final rules (e.g., [18], [7], [6]). We refer to those approaches as ordered replacement and two-phase commit respectively. Unfortunately, they are limited or inefficient, because they focus either only on rule replacements or exclusively on tag-and-match operations.

Ordered replacement cannot always be applied. In Fig. 1, an ordering of rule replacements that preserves both forwarding correctness and the given policy does not exist. In fact, we need to replace rules on u, v, and z, and we cannot assume simultaneous operations, because of uncontrolled factors like different rule installation time across switches [6] or delayed message delivery between the controller and the switches. Thus, we have three cases. If we start from u and $rep(u, f_{u,d})$ is the first operation to be scheduled by the controller, then $f_{u,d}$ is forwarded on path $[u, z, w, d]$ upon $app(rep(u, f_{u,d}))$. This produces a violation of $\mathcal{P}_{u,d}$. Otherwise, if we start from v, then $f_{u,d}$ is forwarded on path $[u, v, w, d]$ upon $app(rep(v, f_{u,d}))$, which also violates the policy. Finally, if $rep(z, f_{u,d})$ is the first operation in the update, packets of $f_{u,d}$ are trapped in a permanent loop between v and z.

Two-phase commit techniques are inefficient. They are based on applying tagging and matching operations on all switches in the network, hence doubling memory utilization at each switch. This comes with two possible consequences. First, the technique *may simply not be applicable* if the memory of a single switch (say, u) is fully used, e.g., by rules for other flows or for backup paths [17]. Second, even if the technique is applicable, it generates a huge overhead which can make it impossible to install new rules, e.g., to deal with traffic surges or security attacks during the update.

B. Combining operations is more powerful

The key intuition exploited by FLIP is that we can profitably combine rule replacements, tagging and matching operations


step 1
step 2
step 3
step 4
FLIP sequence:
$tag(v, f_{u,d}, \tau), rep(w, f_{u,d}),$
$match(z, f_{u,d}, \tau, \emptyset)$
$\Rightarrow rep(u, f_{u,d}) \Rightarrow rep(v, f_{u,d}) \Rightarrow rep(z, f_{u,d})$
d
w
packet tag
τ
u
v
z
$\emptyset$

(a) $t < step2$ (b) $step2 \leq t < step3$ (c) $t \geq step3$

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 four additional rules needed by [18], [7].

on different switches. To this end, it builds upon basic properties verified if given operations are applied in a certain order. In the example of Fig. 1, for instance, FLIP detects that 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 loop between v and z after traversing z at most twice. Hence, FLIP produces a safe update in which z matches throughout the process, v starts tagging before any rule replacement, and replacements are carefully ordered.

FLIP hugely reduces the number of added rules. When run on the example in Fig. 1, FLIP's overhead is a single additional rule on z. This is much more efficient (66% additional rule saving) than two-phase commit techniques that would install additional rules on u, v, and z. The operational sequence computed by FLIP 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 send operations in a step to switches only after it is sure that operations in the previous step are applied (e.g., after receiving an acknowledgment from the switches [8]). In contrast, operations in the same step can be sent simultaneously by the controller: This does not mean that they are executed simultaneously; rather, it implies that their relative order does not matter.

FLIP admits correct paths discarded by other approaches. The bottom part of Fig. 2 provides an illustration of the paths followed by packets of $f_{u,d}$ in *any possible state* explored during the application of the FLIP sequence. It visually proves that both packet delivery and policy compliance (i.e., traversal of link (v, z)) are guaranteed. Indeed, packets either follow the initial or final paths (see Fig. 2(a) and 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 path in Fig. 2(b) would have been discarded by ordered replacement techniques, since it contains a loop.

FLIP efficiently supports strong consistency, that is, policies imposing that either the initial or the final paths have to be kept for every flow throughout the update. Even in this case, FLIP generally uses fewer rules than two-phase commit techniques. In Fig. 1, for instance, it duplicates only on v and z, achieving a 33% rule saving. Indeed, packets in $f_{u,d}$ are forwarded on either the initial or the final path if v and z apply the initial or final rule consistently with u. This can be ensured if v and z match and u tags throughout the update, so that v and z apply their respective final rules when u sets a given tag θ_f to signal that it is in its final state.

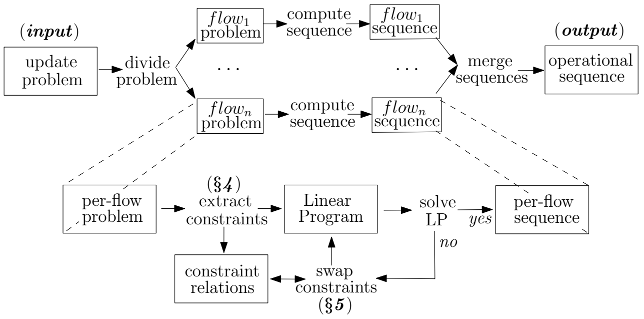


Fig. 3. High-level view of FLIP algorithm. Non-boxed text are used for FLIP internal procedures, and text boxes for the corresponding input and output.

III. FLIP Overview

Fig. 3 overviews FLIP. We now describe FLIP's input (§III-A), output (§III-B), and algorithmic core (§III-C). Since we publicly released a FLIP implementation [21], 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 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 u and the destination address d. Hence, packets sourced at u and destined to d belong to the same flow $f_{u,d}$. Each flow is associated to a destination to which packets have to be delivered and a set of sources, i.e., switches attached to the origin of the packets. We define *forwarding paths* for a flow f as the network paths $[s_0, s_1, s_2, \ldots, d]$, where s_0 is a source, each s_i is a switch, and d is the destination. We admit equal-cost multipath (ECMP), implying that multiple forwarding paths can exist 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 eventually 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. A *blackhole* occurs when a forwarding path is $[s, \ldots, b]$ terminates in a switch b different from the destination and without a rule to forward the packet further. An *evil loop* can occur when packets of a given flow are bounced back and forth indefinitely, among a finite number of switches. In other words, a forwarding path is infinite. Note that the loop in Fig. 2(b) is not evil since the forwarding path used for $f_{u,d}$, i.e., $[u, z, v, z, w, d]$, 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 generated during the update. With respect to previous works that either support strong consistency [6], [18] or single-node traversal [11], FLIP supports a larger variety of practical policies. Supported policies indeed include traversal of single nodes or links (e.g., for firewalling [18]), but also of sub-paths (e.g., for distributed middleboxing [16], service chaining [4] or QoS-based traffic engineering [1]). Generalizing the notation in Fig. 1, we indeed 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 one among 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 we want a given flow 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. This 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 input property preservation independently of the relative order in which they are actually applied by switches, hence they can be sent by the controller in any order or in parallel; and (iii) no operation of a group G_{i+1} can be executed before any operation in G_i. We refer to G_i as i-th *update step*.

To achieve maximum robustness, we assume that messages between the controller and switches can be subject to an arbitrary large but finite delay (they will be retransmitted if lost), and that a switch can take a non-deterministic time [6] to apply an operation once the message has been received. This implies that the operational sequence produced by FLIP does not rely on the simultaneous application of multiple operations.

C. Algorithmic Overview

At a high-level, FLIP adopts a *divide-and-conquer* approach (see 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 tackling 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 of the per-flow sequences to be merged as a set of dependencies and use a scheduling algorithm as in [6] to optimize update speed.

Computation of policy-preserving per-flow sequences is the most novel part of FLIP. It is based on two core procedures.

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 ensure a safe update (if satisfied). We distinguish between replacement and tag-and-match constraints. A *replacement constraint* imposes a certain ordering between rule replacements. A *tag-and-match constraint* imposes that some switches have to tag packets consistently with the applied rule (initial or final) and another switch has to match those tags during the update. For example, to avoid the loop between v and z in Fig. 1, the replacement constraint generated by FLIP is $app(rep(v, f_{u,d})) < app(rep(z, f_{u,d}))$. The corresponding tag-and-match constraint imposes that z matches throughout the update. To setup packet tagging and matching, it requires that $tag(v, f_{u,d}, \tau), match(z, f_{u,d}, \tau, \emptyset)$ are in the first update step G_1. Moreover, to stop matching only at the end of the update, it mandates $app(rep(z, f_{u,d})) > app(rep(n, f_{u,d}))$ for any non-matching switch n.

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 *active*. FLIP tries to compute a solution that satisfies all active constraints by translating the set of active constraints into a linear program (LP) where the objective function is to minimize the number of update steps. FLIP then 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). Since a matching constraint is always satisfiable,

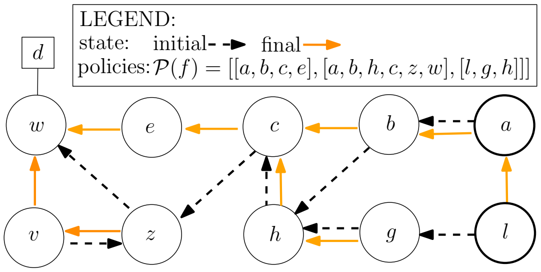

LEGEND:
state: initial
final
policies: $\mathcal{P}(f) = [[a, b, c, e], [a, b, h, c, z, w], [l, g, h]]$
d
w
e
c
b
a
v
z
h
g
l

Fig. 4. An instance of our update problem.

FLIP eventually reaches a combination of active matching and replacement constraints for which a solution exists.

In the following sections, we provide more details on both constraint extraction and swapping.

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 a node n are the nodes 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, with $\sigma \in \{init, fin\}$, we define crucial predecessors of n for f in σ as a set C of nodes such that for every forwarding path $Q = [s \dots n \dots d]$ in σ (i) Q can be written as $[s \dots p, m \dots n \dots d]$ with $p \in C$ and m next-hop of p in σ (possibly,$m = n$); and (ii) m is not a next-hop of p in any forwarding path for f in the state $\{init, fin\} \setminus \{\sigma\}$. Crucial predecessors are initial if $\sigma = init$, and final otherwise. In Fig. 4, a set of initial crucial predecessors of w for flow f is $\{z\}$. Indeed, all initial paths $[s \dots w, d]$, with $s \in \{l, a\}$, can be rewritten as $[s \dots z, w, d]$ 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 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 uses the same next-hop in both the initial and the final states for f. As a result, $cpreds(w, f, init) = \{z\}$ in Fig. 4. 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 n; 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. In contrast, 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 on them 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 [20]: 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 [20]. 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 subject to an input policy, FLIP separately colors G^i_f and G^t_f. We then generate 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.

Node coloring. Given any graph G, with $G = G^i_f$ or $G = G^t_f$, colors are assigned by FLIP using the following algorithm. First, it 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

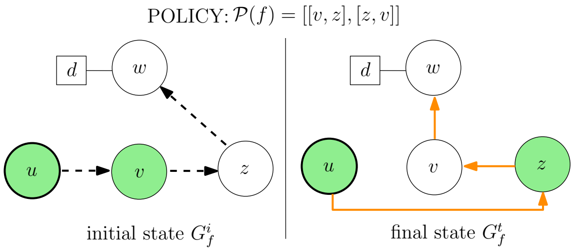


Fig. 5. Coloring for the update scenario in Fig. 1.

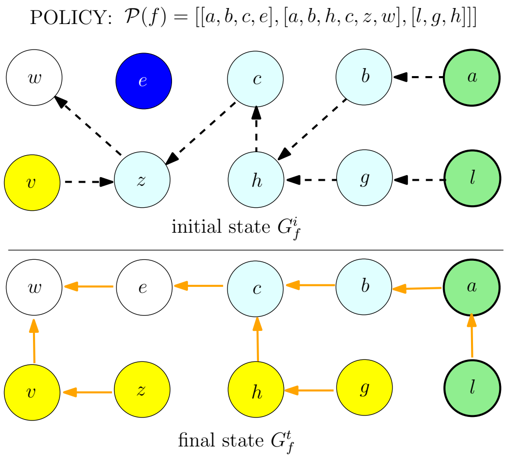


Fig. 6. Coloring for the update scenario in Fig. 4.

	green	cyan	white, yellow
green	-	n>cpreds(n,f,G_f^t) match on n	n>cpreds(n,f,G_f^t) match on n
cyan	n<cpreds(n,f,G_f^i) match on n	*enum*	n>cpreds(n,f,G_f^t) match on n
white, yellow	n<cpreds(n,f,G_f^i) match on n	n<cpreds(n,f,G_f^i) match on n	-

n=analyzed node, f=flow, G_f^i=initial state, G_f^t=final state

TABLE I
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.

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 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 u and v as green in the initial graph, while z and w are 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 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. Consistently, Fig. 6 shows that g, h, b, c and z are cyan in G_f^i for the example in Fig. 4.

Constraint extraction from colored graphs. Starting from node-colored graphs, FLIP extracts multiple sets of constraints for $\mathcal{P}(f)$, according to Table I. In the table, we use expressions like $n < cpreds(n, f, S)$ instead of $\forall p \in cpreds(n, f, S)$ $app(rep(n, f)) < app(rep(p, f))$ for brevity.

Table I shows that FLIP does not generate constraints for nodes which are either (i) green in both G_f^i and G_f^t, or (ii) white in both G_f^i and G_f^t. The rationale is that those nodes cannot be responsible for possible policy violations. Consider a node g which is green in both G_f^i and G_f^t. 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 policy violations. The same applies to any node w which are white in both G_f^i and G_f^t, since $\mathcal{P}(f)$ has to be satisfied before reaching w in both the initial and final state.

In contrast, some constraints are needed for nodes with different colors in G_f^i and G_f^t. Consider, for example, any node r which is white in G_f^i and green in G_f^t, like z in Fig. 5. A rule replacement on r can induce a policy violation from a given source s in G_f^i, as the initial policy path can be bypassed via the final path from s to r (e.g., $[u, z]$, and the final policy path can be circumvented with the initial path from r to the destination of the flow (e.g., $[z, w, d]$). To prevent this case, we constrain the rule replacement on r to be applied before those of all its initial crucial predecessors. This guarantees that no source reaches r with the final path before r uses its final rule. In our example, FLIP adds a replacement constraint $app(rep(z, f)) < app(rep(u, f))$. 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 apply to other combinations of different colors in G_f^i and G_f^t.

Finally, nodes that are cyan in both G_f^i and G_f^t (like b and c in Fig. 6) have to be treated differently. In this case, even computing whether constraints are needed is not obvious. Indeed, there is no simple condition to check whether switches in the middle of a policy path can be part of paths violating $\mathcal{P}(f)$ in intermediate states, since it depends on possible next-hops of both their respective predecessors and successors. For those nodes, FLIP enumerates paths in $G_f^i \cup G_f^t$ that contain at least one node which is cyan in both states. Note that this is a sort of limited path enumeration, which is restricted on the basis of potentially-dangerous nodes (cyan in both states) due to complex policy paths (with more than two nodes). FLIP then pinpoints those among the enumerated paths that violate $\mathcal{P}(f)$. This way, it detects that $[a, b, c, z, w, d]$ is a possible forwarding path for f which violates $\mathcal{P}(f)$ 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 no policy path in $\mathcal{P}(f)$ starts with $[a, b, c, z]$. In particular, FLIP constrains c's rule replacement to be applied before its crucial predecessor on V, that is b in our 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_f^i and G_f^t 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 reported 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 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 in general, for every potential blackhole, loop and policy violation, the different constraints generated by FLIP are stored as alternative.

FLIP tracks dependencies between constraints generated by different violations and involving the same nodes. For example, consider again Fig. 1, and assume that the policy is to preserve strong consistency, i.e., ensure that either the initial path $[u, v, z, w]$ or the final one $[u, z, v, w]$ is followed. A tag-and-match constraint in which v tags and z matches still avoids the evil loop between v and z. However, if we match on z we must also match on v to avoid paths different from both the initial and final ones (like the one in Fig. 2(b)). Hence, FLIP stores the tag-and-match constraint which matches on v as dependent on the tag-and-match constraint on z. Such a dependency is identified during the enum procedure in the policy constraint extraction (see Table I), when considering nodes cyan in both the initial and final state, and involved in the same loop (like v and z in the example just discussed).

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 is tailored to efficiently find a safe sequence with few matching operations, to limit the memory overhead of the update.

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 is complete**. Indeed, FLIP always finds a solution to its input problems since it falls back to the always-applicable two-phase commit approach [18] in the worst case.

At each invocation of the constraint swapping procedure, we select constraints to be swapped using a heuristic targeted to quickly find a solvable set of constraints. Indeed, 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 [2], a minimal set of active constraints that cannot be satisfied simultaneously; and (ii) M has the minimal number of dependent constraints among alternatives for R.

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. First, it removes any replacement constraint R' that has M as alternative from the set of active constraints: Indeed, the potential incorrectnesses that R' is meant to avoid are now prevented by M. Second, FLIP adds all M's dependencies to the set of active constraints, i.e., respecting the definition of dependent constraints (see §IV). Third, FLIP rewrites existing replacement constraints. Let r be the switch matching in M. For each replacement constraint $C = app(rep(x, f)) < app(rep(r, f))$, FLIP replaces C with a set of constraints $C'_i = app(rep(x, f)) < app(rep(y, f))$ for every crucial predecessor y of r. This is needed to preserve the semantics of C after we have decided that switch r will have to match. Indeed, applying a matching operation to r implies that r uses its initial or final rule depending on the tag in the traversing packets. Since the original intent for constraint C was to prevent r from using its final rule if x was still using its initial one, we need to transfer C on the crucial predecessors of r, which add tags to packets. With the rewritten constraints C'_i, we indeed impose that the final rule is installed on x before one of the crucial predecessors of r installs its final rule (and adding final tags), therefore indirectly forcing r to use its final rule too. We apply a similar rewriting for constraints $app(rep(x, f)) > app(rep(r, f))$.

An example of constraint swapping for the case in Fig. 1 is reported in Fig. 7. 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, all and only replacement constraints are active. This translates to an infeasible LP, where r_x stands for $app(rep(x, f_{u,d}))$, with $x = u, v, z$. The swapping procedure selects $app(rep(v, f_{u,d})) < app(rep(z, f_{u,d}))$ 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, $app(rep(z, f_{u,d})) < app(rep(u, f_{u,d}))$ 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

constraints **LP**

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

$$\boxed{\begin{array}{l} \min\ r_u + r_v + r_z + r_w \\ r_v < r_z \\ r_u < r_v \\ r_z < r_u \\ r_u, r_v, r_z, r_w \text{ integer} \end{array}}$$

swap $app(rep(v, f_{u,d})) < app(rep(z, f_{u,d}))$
with *match on z*

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

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

Fig. 7. A constraint swapping solving the scenario in Fig. 1.

match-and-tag constraints on z, as discussed in §III-C. Note that *match on z* also implies other constraints, imposing that $tag(v, f_{u,d}, \tau)$ and $match(z, f_{u,d}, \tau, \emptyset)$ have to be in the first update step. Since they do not impose constraints on any other operation, FLIP does not include them into the LP but post-processes the LP solution by simply adding those operations to the very first element of the returned sequence.

VI. Evaluation

We evaluate FLIP by performing $50,000$ experiments. In each experiment, we generate an update problem on which we run our FLIP implementation, available at [21]. 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).

As dataset, we use the publicly-available Rocketfuel topologies [19]. 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 node to the destination in the original topology are taken as the initial state. To simulate significant forwarding changes, we then pick 80% of the links at random, and replace their weight with a value chosen uniformly at random among the weights of the original topology. The ECMP shortest paths in the reweighted graph constitute the final state. Finally, we add random policies so that every path from a source to the destination is compliant with at least one policy. We choose non-trivial policies composed of paths longer than 2 nodes, to show FLIP's support for more complex policies than single-node traversal ones considered by [11].

FLIP always computes safe updates and 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 techniques based on ordered rule replacement, e.g., [13]. Those techniques can preserve policies only 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. 8(a). They show that ordered replacement techniques cannot find an operational sequence in more than $\approx 25\%$ of the experiments on any topology. 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 guarantee the input policy (rather than strong consistency) and combining rule replacements with match-and-tag operations (rather than restricting to the former ones).

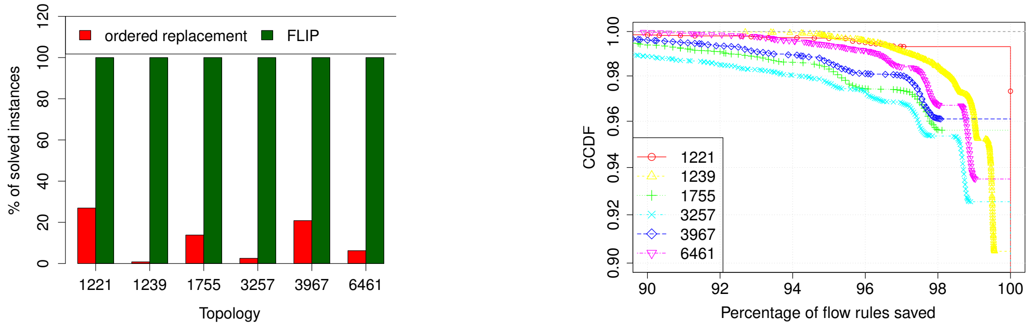
FLIP hugely reduces the number of added rules. The 99.9th percentile of additional rules is 8.7% of the total number of rules, that is, one rule has to be added for 8.7% of the nodes. We now compare FLIP's overhead with the one of two-phase commit techniques [18]. For each experiment, we compute the number N_{DUP} of additional rules added by [18], the number N_{FLIP} of additional rules added by FLIP. We compare those numbers, calculating the percentage of saved rules as $\frac{N_{DUP}-N_{FLIP}}{N_{DUP}} \times 100$. To be fair, we assumed that the two-phase commit approach does not match on nodes with the same next-hops in the initial and final states, as suggested in [18] to reduce the number of additional rules. Fig. 8(b) shows the Cumulative CDF of the results. A data point (x, y) on the plot indicates that, for a fraction y of the experiments, FLIP saved at least x% of the flow rules that would be used by [18]. Across all topologies, the figure highlights that *in 98% of the experiments* ($y = 0.98$) *FLIP saves at least 94%* ($x = 94$) *of the rules added by [18].* Across all our experiments, at least 87.8% of the rules are saved by FLIP.

FLIP's savings are fundamentally different from those of previous variants of two-phase commit techniques. Prominently, [7] 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. Moreover, it degenerates to [18] in our experiments, since a single flow is updated in them.

FLIP computes fast updates. 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 significantly across 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.

FLIP often terminates in sub-seconds. FLIP's median execution time across all our experiments is 0.176 seconds when run on a commodity server (8-core 2.66GHz CPU[1] and 16 GB of RAM). Moreover, 94% of the instances are solved

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



(a) Success rate of FLIP and of ordered replacement techniques like [13].

(b) Additional rules that FLIP saves relatively to two-phase commit techniques like [18], [6].

Fig. 8. Comparison between FLIP and state-of-the-art approaches.

in less than 1 second, and 99% in less than 4 seconds. The topology showing the worst performance is 1239, the largest one, where the 95th percentile of the execution time was 3.38 seconds and the 99th was 15 seconds. Those results show that FLIP readily supports realistic SDN-update scenarios, ranging from accommodation of new policy changes to online traffic engineering (typically performed at the timescale of few minutes [4]) and pre-computation of failure reaction.

VII. Conclusions

In this paper, we present FLIP, an algorithm to compute operational sequences for safe updates of SDN networks. By design, FLIP enables updates that preserve both forwarding correctness and forwarding policies. Thanks to its novel way to systematically combine rule replacements and additions, FLIP's updates are fast and lightweight. Indeed, as shown by our extensive simulations, they tend to terminate in a very limited number of steps and with minimal overhead on the memory of the switches. Our evaluation also shows that FLIP outperforms previous approaches: In our experiments, it saves more than 90% of the additional rules needed by two-phase commit techniques, and supports 90% more update scenarios than ordered replacement ones.

The model that FLIP uses to reason about the dualism between rule replacement and additions makes FLIP extensible. For instance, FLIP can easily support domain-specific constraints such as memory restrictions on specific switches. We indeed 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. As future work, we plan to provide better support for domain-specific constraints, and to investigate FLIP's extensions for additional types of operations (e.g., time-based rule modifications [15]).

Acknowledgements

This work has been partially supported by ARC grant 13/18-054 from Communauté française de Belgique.

References

[1] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou. A Roadmap for Traffic Engineering in SDN-OpenFlow Networks. *Computer Network*, 71:1–30, 2014.

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

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

[4] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM*, 2015.

[5] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.

[6] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

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

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

[9] 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.

[10] H. Liu, X. Wu, M. Zhang, L.Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

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

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

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

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

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

[16] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[17] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN*, 2013.

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

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

[20] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless Network-Wide IGP Migrations. In *SIGCOMM*, 2011.

[21] S. Vissicchio. FLIP Web site. http://inl.info.ucl.ac.be/softwares/flip.

[22] S. Vissicchio, L. Cittadini, O. Bonaventure, G. G. Xie, and L. Vanbever. On the Co-Existence of Distributed and Centralized Routing Control-Planes. In *INFOCOM*, 2015.