Automata-Theoretic Approach to Verification of MPLS Networks under Link Failures

Ingo van Duijn[1] Peter Gjøl Jensen[1] Jesper Stenbjerg Jensen[1] Troels Beck Krøgh[1] Jonas Sand Madsen[1]
Stefan Schmid[2] Jiří Srba[1] Marc Tom Thorgersen[1]
[1] Aalborg University, Denmark [2] Faculty of Computer Science, University of Vienna, Austria

***Abstract*—Future communication networks are expected to be highly automated, disburdening human operators of their most complex tasks. While the first powerful and automated network analysis tools are emerging, existing tools provide only limited and inefficient support of reasoning about *failure scenarios*. We present P-Rex, a fast *what-if analysis* tool, that allows us to test important reachability and policy-compliance properties even under an *arbitrary number* of failures and in *polynomial-time*, i.e., without enumerating all failure scenarios (the usual approach today, if supported at all). P-Rex targets networks based on Multiprotocol Label Switching (MPLS) and its Segment Routing (SR) extension which feature fast rerouting mechanisms with label stacks. In particular, P-Rex allows to reason about recursive backup tunnels, by supporting potentially infinite state spaces. As P-Rex directly operates on the actual dataplane configuration, i.e., forwarding tables, it is well-suited for debugging. Our tool comes with an expressive query language based on regular expressions. We also report on an industrial case study and demonstrate that our tool can perform what-if reachability analyses on average in about 5 seconds for a 24-router network with over 250,000 MPLS forwarding rules. This is a significant improvement to an earlier prototype of our tool presented in the conference version of our paper where the verification took on average about 1 hour.**

***Index Terms*—Network Verification, MPLS, Prefix Rewriting Systems.**

I. Introduction

Ensuring policy compliance under failures is a challenging task which can quickly overstrain human operators, even of small networks. This is worrisome as already a single link failure can lead to undesirable network behaviors, which is easily overlooked, such as datacenter traffic leaking to the Internet in unintended ways [1]. More generally, unintended behavior after failures can harm the availability, security, and performance of a network [2]. The possibility of *multiple* link failures [3, 4, 5], e.g., due to shared risk link groups [6, 7], exacerbates the problem.

Automation is an attractive alternative to today's manual and error-prone approach to operate communication networks, allowing to overcome the shortcomings of current "fix it when it breaks" approach. Accordingly, over the last years, many powerful tools have been developed to specify and verify communication networks, e.g. [2, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Existing tools usually allow to query various kinds of reachability properties in the network, also accounting for the header fields in the packet and their transformations along the route.

However, while automation allows to overcome the drawbacks of manual network operations, verifying network configurations can still be a complex task, even for a computer: many existing tools have a super-polynomial runtime [10], in the worst case, and some queries are even undecidable [25, 26]. What is more, existing tools that operate on the data plane level do not often provide much support for reasoning about network behavior *under failures*, a major concern of operators responsible for the availability of the network. The existing data plane aproaches to what-if analysis accounting for failures often resort to enumerating all possible failures scenarios, introducing a combinatorial complexity. This may even appear unavoidable: in an n-node network with k failed links, it may seem that all $\binom{n}{k}$ possible failure scenarios need to be examined to verify whether a certain network property (e.g., related to reachability or policy-compliance) holds. Recently there has been a succesful effort to remedy this issue by instead analyzing the control plane while considering multiple data planes at the same time [13, 14], however, these methods are not yet directly applicable to networks based on Multiprotocol Label Switching (MPLS), which are the focus of our paper. The main reason is that in MPLS networks we need to deal with repeated nesting of MPLS labels, which can be in existing tools simulated by creating finite products of labels, however, at the expense of exponential explosion (see e.g. the HSA experiments in Section V).

We are interested in the *fast* and automated verification of MPLS networks, even under failures. MPLS networks are widely deployed today, e.g., used by telcos for traffic engineering or for VPNs, carrying IP and Virtual Private LAN traffic accordingly. MPLS avoids complex routing table lookups by forwarding packets based on short *path labels* (identifying virtual links), rather than long network addresses. Such labels can be accumulated in a *label stack*, e.g., during local Fast Re-Routing (FRR): when a source to a link detects the failure, it will reroute packets through the backup tunnel, by pushing a label onto the stack. This operation can be performed recursively, in case of multiple link failures, and hence introduces a key difference to many other network protocols: a verification tool needs to be able to deal with dynamic header sizes and potentially infinite state spaces.

A. Our Contributions

This paper makes the case for an automata-theoretic approach to the efficient verification of MPLS-based communication networks, allowing to reason about an arbitrary number of link failures and a potentially infinite state space (label stack). In particular, we present P-REX[1], a *what-if analysis* tool which allows to test a wide range of important network properties in *polynomial-time*, independently of the number of failures. The runtime of other existing tools is proportional to the number of failure scenarios which is exponential in the number of failures.

At the core of P-REX lies a powerful yet simple query language based on *regular expressions*, both to specify *packet headers* as well as *paths*. Specifically, queries are of the form

$$< a > b < c > \; k$$

where a and c are regular expressions describing the (potentially infinite) set of allowed initial resp. final headers of packets in the trace, b is a regular expression defining the (potentially infinite) set of allowed sequences of links between pairs of routers, and k is a number specifying the maximum allowed number of failed links. P-REX allows to test properties such as waypoint enforcement (e.g., is the traffic always forwarded through an intrusion detection system) or avoidance of certain countries (e.g., never route via Iceland). P-REX operates directly on the router tables, which has the advantage that it also accounts for possible external changes, e.g., manual changes through the CLI, bugs, or changes made by additional protocols. Moreover, dataplane-based approaches can also reveal problems in the network operation introduced by the algorithms that generate the routing tables from the control plane specifications. P-REX further allows us to account for more complex *traffic engineering* aspects, such as load-balancing, by supporting nondeterminism, as well as more complex *multi-operation chains* modelling aspects of Segment Routing (SR). We prove that checking for query satisfability is an NP-complete problem and in order to deal with this complexity, we present over-approximation and under-approximation techniques to improve the verification speed.

Our experiments demonstrate a convincing performance of P-REX compared to existing tools, on different workloads. For this comparison, we modified the HSA tool [18] in order to be applicable to MPLS-like networks. We also report on an industrial case study and show that P-REX can solve most of the complex queries in the operator's 24-router network containing over 250,000 forwarding table entries in a matter of seconds or minutes in the worst case.

B. *Overview of* P-REX

In a nutshell, given the network configuration, the routing tables, as well as the query, P-REX constructs a pushdown automaton (PDA). The initial header and final header regular expressions of the query are each converted to first a Nondeterministic Finite Automaton (NFA) and then to a Pushdown Automaton (PDA). The path query is converted to an NFA, which is used to augment the PDA constructed based on the network model (using either the over- or under-approximation approach). The three PDAs are combined into a single PDA which we give to the PDA reachability tool Moped [27]. Moped then either provides a trace through the pushdown which witnesses the query, or says that no such witness exists.

	P-REX	NetKAT	HSA	VeriFlow	Anteater
Protocol Support	SR/MPLS	OF	Agn.	OF	Agn.
Approach	Autom.	Alg.	Geom.	Tries	SAT
Complexity	Polynom.	PSPACE	Polynom.	NP	NP
Static	✓	✓	✓	χ	✓
Reachability	✓	✓	✓	✓	✓
Loop Queries	✓	✓	✓	✓	✓
What-if	✓	N/A	✓	N/A	χ
Unlim. Header	✓	N/A	χ	χ	N/A
Performance	✓	✓ [10]	✓	✓	✓
Waypointing	✓	✓	✓	✓	χ
Language	C++	OCaml	Py., C	Py.	C++, Ruby

TABLE I: Comparsion of related tools

At the heart of P-REX lies a novel method for combining an NFA, generated from the query, and a pushdown automaton, into a single PDA which then simulates the two automata running in lockstep. This method restricts the paths through the PDA emulating the MPLS behavior. Our tool includes several optimizations to further improve the performance, such as "top of stack reduction", which safely calculates which labels can be at the top of stack in a given PDA state, allowing us to reduce the amount of transitions in the PDA.

II. RELATED WORK

Motivated by the observation that many recent network outages were caused by human errors, e.g., [1, 28, 29], we witness major research efforts toward more automated and formal approaches to operate and verify networks [2, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 30].

Typically, network verification tools are given some model or configuration of the control plane (e.g. [2, 12, 14, 15, 16]) or the data plane (e.g. [12, 17, 18, 23, 31]), and some properties to check. Table I provides an overview and comparison of several selected tools: Some tools are specific to a certain protocol, such as BGP [32] or OpenFlow (OF), others are protocol agnostic (agn.) [18]. Some tools rely on automata-theoretic approaches (autom.), others on algebra (alg.), geometric techniques (geom.), or SAT/SMT solvers. Some tools only support basic reachability queries, others support loop-detection and waypointing. Further interesting works consider randomized approaches [33, 34].

To just give some examples, NetKAT [10] focuses on static verification of the network configuration and allows checking for failures in terms of reachability and forwarding loops, with a support for waypointing. NetKAT sets itself apart from our, and other tools, particularly in its approach to modeling and expressing the network. Header Space Analysis (HSA) [18] is also a static verification tool. As the name suggests, this tool is focused on utilizing the headers of packets for the verification. HSA only covers basic reachability and forwarding loops properties, but not more complex queries. Unlike NetKAT, HSA generates a geometric model from the packet headers and

[1] P-REX stands for Pushdown analysis tool with REgular eXpressions.

the network configuration. Headers are abstract in that their protocol-specific meanings are ignored. The tool developed in our paper removes the restriction on header sizes being bounded (and by representing them symbolically as pushdown automata, it achieves an exponential speedup). VeriFlow [23] focuses on being able to detect bugs. This tool is effectively added to the networks configuration and acting as a layer between the network and an SDN controller. VeriFlow models data-plane information as boolean expressions and uses a SAT solver algorithm to check for failures. Anteater [22] is similar to VeriFlow in that it converts the data plane information to boolean functions and uses a SAT solver to check whether the invariants are violated. Anteater focuses mainly on detecting reachability, forwarding loops, and packet loss as invariants.

Thanks to these research efforts, today we have a fairly good understanding of how to achieve efficient network verification in various contexts. However, much less is known about the verification of network policies *accounting for (possible) failures*. We in this paper are particularly interested in the verification of the data plane, and we refer the reader to Abhashkumar et al. [9] for an overview of control plane solutions. We also note that most existing data plane verification tools can be extended to account for failures by simply generating all possible data planes under failures, which however introduces an exponential runtime [18, 23]. Furthermore, while the design of fast-rerouting mechanisms for the data plane has been an active field of research for many years, see the recent survey by Chiesa et al. [35], we are not aware of any results on the fast verification of MPLS fast reroute.

To the best of our knowledge, P-Rex is the first tool to support a polynomial-time what-if analysis, accounting for all possible failure scenarios and supporting a possibly infinite state space, allowing to analyze recursive backup tunnels and arbirary and dynamic (unbounded) label stacks. A preliminary tool prototype of P-Rex has been presented at CoNEXT 2018 [31], based on the theory developed in [19], and the current paper is an extended version of these two papers. Compared to these early versions, we simplify the query language, present novel top-of-the-stack optimizations, include all proofs as well as an NP-hardness proof motivating the over- and under-approximation, and report on more extensive and conclusive evaluations. We also re-implement P-Rex in C++ (the original version was written in Python) and change the encoding, which allows us to reduce the runtime by orders of magnitude (from hours to seconds in some cases). We make our open-source code and evaluation artifacts available for reproducibility. Finally, we note that P-Rex already led to first followup works. In particular, AalWiNes tool demonstrates that the ideas underlying P-Rex can be extended to verify not only logical but also *quantitative* properties, using a *weighted* automaton approach.

III. Formal Network Model

We shall first present our general model of MPLS-based networks, including the routing tables with priorities and the definition of a network trace. Let L_M be a nonempty set of MPLS labels that appear (possibly arbitrarily nested) in headers of packets of an MPLS network and L_{IP} be a nonempty set of IP-headers. We define the set Ops of allowed MPLS operations on a packet header by $Ops = \{swap(\ell) \mid \ell \in L_M\} \cup \{push(\ell) \mid \ell \in L_M\} \cup \{pop\}$. We use the Kleene star notation S^* to denote the set of sequences of elements from any set S, e.g. Ops^* denotes the set of all (possibly empty) sequences of MPLS operations.

$\vdots$
$\ell_2 \in L_M$
$\ell_1 \in L_M$
$\ell_0 \in L_{IP}$

Fig. 1: A valid label-stack header

Definition 1 (MPLS Network)**.** An MPLS network is a tuple $N = (V, I, E, L, \tau)$ where

- V is a finite set of routers,
- I is the finite set of all global interfaces in the network partitioned into disjoint sets I_v of local interfaces for each router $v \in V$ such that $I = \bigcup_{v \in V} I_v$,
- $E \subseteq I \times I$ is the set of links connecting interfaces that satisfy if $(out, in) \in E$ then $(in, out) \in E$, if $(out, in), (out', in) \in E$ then $out = out'$, and if $(out, in), (out, in') \in E$ then $in = in'$,
- $L = L_M \cup L_{IP}$ is the set of the label stack symbols where L_M is the MPLS label set and L_{IP} is a set of labels for IP routing information where $L_M \cap L_{IP} = \emptyset$, and
- $\tau : E \times L \to (2^{E \times Ops^*})^*$ is the global routing table. For every link $(out, in) \in E$ and a top (left-most) label, it returns a sequence (representing priorities in case of link failures) of traffic engineering groups that contain pairs: an outgoing interface $(out', in') \in E$ s.t. $out', in \in I_v$ for some $v \in V$ and a sequence of MPLS operations $\omega \in Ops^*$ to be performed on the packet header. We can represent the global routing table as a collection of local routing tables with $\tau_v : E_v \times L \to (2^{\overline{E_v} \times Ops^*})^*$ for each router $v \in V$ with $E_v = \{(out, in) \in E \mid in \in I_v\}$ and $\overline{E_v} = \{(out, in) \in E \mid out \in I_v\}$.

We fix a set F where $F \subseteq E$ of failed links between interfaces. A link $e \in E$ is *active* if $e \notin F$.

MPLS networks often tunnel traffic containing some underlying header (typically an IP address) which we assume belongs to the set L_{IP}; the MPLS labels are stacked on top of this label. The structure of a valid label-stack header is illustrated in Figure 1. This is formalized in the following definition where $A \circ B = \{ww' \mid w \in A, w' \in B\}$ for any two sets of strings $A, B \subseteq L^*$.

Definition 2 (Valid Header)**.** For a given network $N = (V, I, E, L, \tau)$ we define the set of valid headers $H = L_M^* \circ L_{IP}$.

The MPLS operations manipulate the label-stack header by switching out the top-most label (left-most symbol in our notation) with another one, pushing a new label or removing a label from the top of the stack. A sequence of such MPLS operations performed on a valid header must ensure that we again obtain a valid header (otherwise the execution of the label-update sequence fails and a packet is dropped). This is formalized in the following definition.

Definition 3 (Header Rewrite Function)**.** A partial *header rewrite function* $\mathcal{H} : H \times Ops^* \hookrightarrow H$ is defined by (where $\omega, \omega' \in Ops^*$, $\ell \in L$, $h \in H \cup \{\epsilon\}$ and ϵ is the empty sequence of labels):

$$\mathcal{H}(\ell h, \omega) =$$
$$\begin{cases} \ell h & \text{if } \omega = \epsilon \\ \mathcal{H}(\ell' h, \omega') & \text{if } \omega = swap(\ell') \circ \omega' \text{ and } \ell' \in L_M \\ \mathcal{H}(\ell' \ell h, \omega') & \text{if } \omega = push(\ell') \circ \omega' \text{ and } \ell' \in L_M \\ \mathcal{H}(h, \omega') & \text{if } \omega = pop \circ \omega' \text{ and } \ell \in L_M \\ undefined & \text{otherwise.} \end{cases}$$

We observe that for any $h \in H$ and any $\omega \in Ops^*$ we always have $\mathcal{H}(h, \omega) \in H$ (provided that $\mathcal{H}(h, \omega)$ is defined). In other words, the header rewrite function preserves the valid structure of the label-stack symbols, otherwise it is undefined. As an example let $L_M = \{10, 20, 30\}$ and $L_{IP} = \{ip\}$. Then $\mathcal{H}(20 \circ 10 \circ ip,\ pop \circ swap(20) \circ push(30) \circ push(10)) = 10 \circ 30 \circ 20 \circ ip$ whereas $\mathcal{H}(20 \circ 10 \circ ip,\ pop \circ swap(30) \circ push(ip))$ is undefined as the expected outcome $ip \circ 30 \circ ip \notin H$ is not a valid header because a stack must contain exactly one IP-label and this should be the bottom (right-most) label.

A. *Network Example*

In Figure 2, we provide an example of a simple network with ten routers $V = \{v_0, \ldots, v_9\}$ and depicted links that have names $e_0, \ldots, e_{10}$. $L = L_M \cup L_{IP}$ consist of

- $L_M = \{10, 11, 20, 21, 22, 30, 31, 40, 41, 101, 102\}$, and
- $L_{IP} = \{ip_{v_8}, ip_{v_9}\}$.

The routing table τ for our example network is given in Table II and there are no rules for the routers v_0, v_1, v_8 and v_9, as they are assumed to belong to another network. Instead of a sequence of sets that τ should return, we give each rule in the table a priority such that all rules with priority 1 form the first traffic engineering group of (high priority) rules in the τ function, and rules with the next priority 2 form the second set of (fast failover) rules. Intuitively, if at least one rule of priority 1 is applicable and can forward the packet to some active link then one such rule will be (nondeterministically) applied. If all output links of rules with priority 1 are inactive (due to failed link or links) then (and only then) we consider the rules with the next priority 2 and so on. The semantics of the network behaviour is given by means of network traces.

B. *Network Traces*

Let us fix a network $N = (V, I, E, L, \tau)$ together with the set of failed links $F \subseteq E$. A *trace* is a routing of a packet in the network that consists of a sequence of active links together with the corresponding label-stack header of the packet during the corresponding hop.

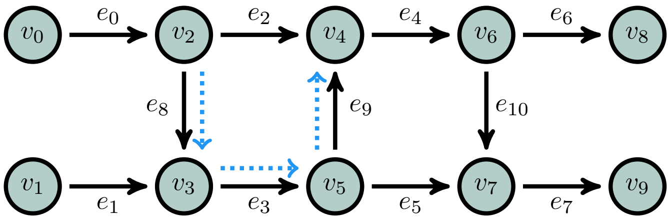

Fig. 2: A network example with a failover tunnel

Router	In E	Label	Priority	Out E	Operation
v_2	e_0	ip_{v_8}	1	e_2	$push(10)$
	e_0	ip_{v_8}	2	e_8	$push(10) \circ push(101)$
	e_0	ip_{v_9}	1	e_2	$push(20)$
	e_0	ip_{v_9}	2	e_8	$push(20) \circ push(101)$
v_3	e_1	ip_{v_8}	1	e_3	$push(30)$
	e_1	ip_{v_9}	1	e_3	$push(40)$
	e_8	101	1	e_3	$swap(102)$
v_4	e_2	10	1	e_4	$swap(11)$
	e_2	20	1	e_4	$swap(21)$
	e_9	10	1	e_4	$swap(11)$
	e_9	20	1	e_4	$swap(21)$
	e_9	31	1	e_4	$swap(11)$
v_5	e_3	30	1	e_9	$swap(31)$
	e_3	40	1	e_5	$swap(41)$
	e_3	102	1	e_9	pop
v_6	e_4	11	1	e_6	pop
	e_4	21	1	e_{10}	$swap(22)$
v_7	e_{10}	22	1	e_7	pop
	e_5	41	1	e_7	pop

TABLE II: Routing table for the network from Figure 2 (routers v_0, v_1, v_8 and v_9 have empty routing tables)

Before we give the formal definition of a trace, we shall fix some notation. Let $\tau(e, \ell) = O_1 O_2 \ldots O_n$ where each $O \in \{O_1, \ldots, O_n\}$ is a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \ldots, (e_m, \omega_m)\}$ consisting of output links and sequences of MPLS operations such that the group O_1 has a higher priority than O_2, and O_2 has a higher priority than O_3 and so on. By $E(O) = \{e_1, e_2, \ldots, e_m\}$ we denote the set of all links that appear in the traffic engineering group O and we call such a group *active* if there is at least one i, $1 \leq i \leq m$, such that the link e_i is active (i.e. $e_i \in E \smallsetminus F$).

Finally, we define a function that returns the set of all active rules in the sequence of a traffic engineering groups as follows: $\mathcal{A}(O_1 O_2 \ldots O_n) = \{(e, \omega) \in O_j \mid e \text{ is an active link}\}$ where j is the lowest index such that O_j is an active traffic engineering group. If no such j exists then $\mathcal{A}(O_1 O_2 \ldots O_n) = \emptyset$.

Definition 4 (Network Trace)**.** A *trace* in a network $N = (V, I, E, L, \tau)$ with the set $F \subseteq E$ of failed links is any finite sequence $(e_1, h_1), \dots, (e_n, h_n)$ of link-header pairs from $(E \setminus F) \times H$ where for all i, $1 \leq i < n$ we have that $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, head(h_i)))$ where $head(h_i)$ is the top (left-most) label of h_i.

The network routing table from Table II encodes four label switched paths from either v_0 or v_1 to either v_8 or v_9, depending on the destination IP address. An example of a trace without any failed links ($F = \emptyset$) follows.

$$(e_0, ip_{v_8}), (e_2, 10 \circ ip_{v_8}), (e_4, 11 \circ ip_{v_8}), (e_6, ip_{v_8})$$

In our example network there is one protected link e_2. In order to protect the link, we create a backup tunnel. In the routing table, the rules for the backup tunnels have a lower priority than the preferred rules with priority 1, so that they are employed only in case of failed links. Hence if for example the link between v_2 and v_4 fails, i.e. $F = \{e_2\}$, then we get the following trace instead

$$(e_0, ip_{v_8}), (e_8, 101 \circ 10 \circ ip_{v_8}), (e_3, 102 \circ 10 \circ ip_{v_8}),$$
$$(e_9, 10 \circ ip_{v_8}), (e_4, 11 \circ ip_{v_8}), (e_6, ip_{v_8})$$

so that the failed link is tunneled through the routers v_3 and v_5 after which the original label switching path is restored.

C. A Query Language for MPLS Networks

We now present our query language for specifying the presence of network traces with certain properties. Assume an MPLS network $N = (V, I, E, L, \tau)$. A reachability *query* in the network N is of the form

$$< a > b < c > k$$

where

- a is a regular expression defining a language over the set of labels L, describing the (potentially infinite) set of allowed initial label-stack headers,
- b is a regular expression defining a language over the set of links E, describing the (potentially infinite) set of allowed routing paths through the network,
- c is a regular expression defining a language over the set of labels L, describing the (potentially infinite) set of label-stack headers at the end of the trace and
- k is a number specifying the maximum allowed number of failed links.

Formally, we assume the following syntax for regular expressions.

Definition 5 (Regular Expression)**.** A regular expression over the alphabet Σ ranged over by the symbols $s, s_1, s_2, \dots$ is given by the abstract syntax

$$\begin{aligned} a ::= \;& \cdot \mid [s_1, s_2, \dots, s_n] \mid [\hat{}\, s_1, s_2, \dots, s_n] \mid \\ & a_1 | a_2 \mid a_1 a_2 \mid a^* \mid a^+ \mid a^? \end{aligned}$$

where the semantics, denoted $Lang(a) \subseteq \Sigma^*$, is given by

$$\begin{aligned} Lang(\cdot) &= \Sigma \\ Lang([s_1, s_2, \dots, s_n]) &= \{s_1, s_2, \dots, s_n\} \\ Lang([\hat{}\, s_1, s_2, \dots, s_n]) &= \Sigma \setminus \{s_1, s_2, \dots, s_n\} \\ Lang(a_1 | a_2) &= Lang(a_1) \cup Lang(a_2) \\ Lang(a_1 a_2) &= Lang(a_1) \circ Lang(a_2) \\ Lang(a^*) &= Lang(a)^* \\ Lang(a^+) &= Lang(a) \circ Lang(a^*) \\ Lang(a^?) &= Lang(a) \cup \{\epsilon\} \end{aligned}$$

The set of all regular expressions over Σ is denoted by $Reg(\Sigma)$ and instead of $[s]$ we write just s for the singleton symbol in the selection construct.

Remark 1 (Negation and Conjunction)**.** We omit the negation and conjunction-operators from the syntax of regular expression to avoid an expensive determinization of the equivalent NFA, however, these operators can be easily added if needed.

We now provide a formal definition of a network query.

Definition 6 (Query)**.** A *query* for a network $N = (V, I, E, L, \tau)$ is an expression $< a > b < c > k$ where $a, c \in Reg(L)$, $b \in Reg(E)$ and $k \geq 0$.

Remark 2 (Addressing elements of E)**.** We syntactically denote elements of $(out, in) \in E$ as $out\#in$ and we extend and abuse this notion over routers, s.t. $v\#v' = \{out\#in \in E \mid out \in I_v \text{ and } in \in I_{v'}\}$. Furthermore, we let $\cdot$ denote "any interface" in this syntax, s.t. $out\#\cdot = \{out\#in \in E \mid in \in I\}$ and $\cdot\#in = \{out\#in \in E \mid out \in I\}$. We extend this dot-notation to routers s.t. $v\#\cdot = \bigcup_{v' \in V} v\#v'$ and $\cdot\#v' = \bigcup_{v \in V} v\#v'$.

Remark 3 (Addressing elements of L)**.** We use the abbreviations:

- **ip** $= [i_0, \dots, i_n]$ where $\{i_0, \dots, i_n\} = L_{IP}$ to stand for "any IP label",
- **mpls** $= [\ell_0, \dots, \ell_m]$ for "any MPLS label", and
- **smpls** $= [\ell_{m+1}, \dots, \ell_n]$ for "any sticky MPLS label" (representing any bottom-of-the-stack MPLS label)

where $\{\ell_0, \dots, \ell_m, \ell_{m+1}, \dots, \ell_n\} = L_M$. We remark that the first MPLS-label on the stack after an IP-address is called a sticky-label, and can be treated differently in the routing-tables, implying that any label comes in both a sticky and non-sticky version. We denote by L_M both sticky an non-sticky labels and let **mpls** and **smpls** yield a partition of L_M. In Table 2 labels below 100 are always used as sticky labels and labels above 100 are non-sticky. Moreover, as we are interested only in MPLS routing in this paper, we may use the abbreviation **ip** in our queries, however, mentioning concrete IP-labels is not allowed.

Finally, we define when a network trace satisfies a query.

Definition 7. A trace $(e_1, h_1), (e_2, h_2), \dots, (e_n, h_n)$ in a network $N = (V, I, E, L, \tau)$ with the set F of failed links *satisfies* a query $< a > b < c > \ k$ if and only if $|F| \leq k$, $h_1 \in Lang(a)$, $h_n \in Lang(c)$ and $e_1 e_2 \ldots e_n \in Lang(b)$.

The decision problem we want to solve is defined as follows.

Problem 1 (Query Satisfiability Problem)**.** Given a network and a query $q \ = < a > b < c > \ k$ is there a trace in the network with at most k failed links that satisfies q?

Considering the network from our running example defined by the routing table in Table 2, we can notice that the query

$$< \mathbf{ip} > [\cdot\#v_2] \ \cdot^* \ [\cdot\#v_8] < \mathbf{ip} > \ 0$$

is satisfied due to the existence of a trace that starts with the initial header ip_{v_8} arriving on e_0 to v_2 and reaches in a number of hops the router v_8 with the header ip_{v_8}. The trace is possible without any failed links by visiting the routers v_2, v_4, v_6 and v_8 as demonstrated in Section III-B. On the other hand the query

$$< \mathbf{ip} > [\cdot\#v_2] \ \cdot^* \ [\cdot\#v_5] \ \cdot^* \ [\cdot\#v_8] < \mathbf{ip} > \quad 0$$

is not satisfied, as without any failed links the traffic arriving at v_2 from the link e_0 with any ip-header is never routed though the router v_5, however, the same query which allows one link failure is satisfied as shown e.g. by the second trace in Section III-B. Another query

$$< \mathbf{ip} > [\cdot\#v_2] \, [\hat{} \cdot \#v_4]^* \, [\cdot\#v_8] < \cdot^* > \ 1$$

asks whether, under the assumption that at most one link fails, a packet with an ip-header can, from the router v_2 (received via any link), reach (with arbitrary header) the router v_8 while avoiding the router v_4. This query is not satisfied in our example network because the router v_4 cannot be avoided neither in the the routing without any failed links, nor when using the backup tunnel should the link e_2 fail.

As a last example, the query

$$< \mathbf{mpls\ smpls\ ip} > [\cdot\#v_3] \, \cdot^* \, [\cdot\#v_4] < \mathbf{smpls\ ip} > \ 0$$

specifies the initial header with two MPLS labels on top of the IP label, one sticky and one non-sticky. This imitates that a fast re-routing happened prior to v_3. The query then asks whether the fast re-route tunnel can end in v_4 if no further link fails, indicated by the end header specified as $\mathbf{smpls\ ip}$. The given query is satisfied for the running example as v_3 can swap label 101 (which is included in $\mathbf{mpls}$) with label 102. The packet is then forwarded to v_5 where label 102 is popped, forwarding the packet to v_4, leaving the header exactly at $\mathbf{smpls\ ip}$.

D. NP-Hardness Result

We shall now prove that the problem whether a given query is satisfied in an MPLS network is NP-hard (and in fact NP-complete as we can guess the set of failed links and in polynomial time verify whether the query under the given set of failed links is satisfied—see Remark 4). The NP-completeness of the query satisfiability problem justifies the need for the design of over- and under-approximation techniques (described in the following section) in order to achieve worst-case polynomial time verification algorithms.

Router	In E	Label	Priority	Out E	Operation
v	b	g_1	1	x_1	swap$(c_1) \circ$ push(g)
	b	g_2	1	x_2	swap$(c_1) \circ$ push(g)
	b	g_2	1	x_2	swap$(c_2) \circ$ push(g)
	b	g_3	1	x_3	swap$(c_1) \circ$ push(g)
	b	g_3	2	$\overline{x_3}$	swap$(c_2) \circ$ push(g)
	x_1	g	1	b	$swap(g_1)$
	x_1	g	1	b	$swap(g_2)$
	x_1	g	1	b	$swap(g_3)$
	x_2	g	1	b	$swap(g_1)$
	x_2	g	1	b	$swap(g_2)$
	x_2	g	1	b	$swap(g_3)$
	x_3	g	1	b	$swap(g_1)$
	x_3	g	1	b	$swap(g_2)$
	x_3	g	1	b	$swap(g_3)$
	$\overline{x_3}$	g	1	b	$swap(g_1)$
	$\overline{x_3}$	g	1	b	$swap(g_2)$
	$\overline{x_3}$	g	1	b	$swap(g_3)$

TABLE III: Routing table for $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})$

Theorem 1. Query satisfiability problem is NP-hard.

Proof. By polynomial time reduction from CNF SAT [36]. Consider a Boolean formula $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ in conjunctive normal form with m clauses $C_1, ..., C_m$ that are formed by disjunctions of literals over k variables $x_1, ..., x_k$. We let $x_i \in C_j$ denote that the literal x_i occurs positively in the clause C_j and $\overline{x_i} \in C_j$ denotes that the negation of x_i occurs in C_j. For the given ϕ, we construct an instance of query satisfiability problem such that the query is satisfied with at most k links failing if and only if ϕ is satisfiable.

We construct one router v with $2k + 1$ self-looping links named x_i and $\overline{x_i}$ for all i, $1 \leq i \leq k$, and a self-loop "back" link b. The valuation of a variable x_i is encoded in the failure of the corresponding link x_i (if the link is not failed then the valuation of x_i is true and if the link is failed the valuation of x_i is false). Furthermore, let $L_{IP} = \{\perp\}$ and $L_M = \{g, g_1, ..., g_k, c_1, ..., c_m\}$ be the MPLS labels we use on the stack.

We define $\tau(b, g_i) = O_1 O_2$ where

$$\begin{aligned} O_1 = &\quad \{(x_i, swap(c_j) \circ push(g)) \mid x_i \in C_j\} \\ O_2 = &\quad \{(\overline{x_i}, swap(c_j) \circ push(g)) \mid \overline{x_i} \in C_j\} \ . \end{aligned}$$

This should be read as: when routing from the backlink with label g_i on top of the stack, it is possible to route to link x_i (resp. $\overline{x_i}$) while pushing a label c_j for any clause C_j that contains the literal x_i (resp. $\overline{x_i}$), corresponding to a clause that is satisfied by x_i (resp. $\overline{x_i}$). Additionally, when routing to this next link, another temporary label g is pushed on top of the stack. Next, we need to be able to route back to link b and nondeterministically replace the label g with a label g_i that corresponds to a variable x_i that satisfies the next clause. Therefore, we add the following two rules to τ for each variable link x_i and $\overline{x_i}$:

$$\tau(x_i, g) = \qquad \{(b, swap(g_h)) \mid 1 \leq h \leq k\}$$
$$\tau(\overline{x_i}, g) = \qquad \{(b, swap(g_h)) \mid 1 \leq h \leq k\} \ .$$

Thus, a trace through this network starting on link b can be interpreted as successively considering a variable, and then pushing a label on top of the stack whose corresponding clause is satisfied by the valuation of the considered variable. Note that the same variable can be considered multiple times because the same variable can satisfy multiple clauses, but since the set of failed links is fixed, so is the valuation of that variable. Table III contains an example of this construction. To complete the definition of query satisfability problem, we define the query

$$< [g_1, ..., g_k]\bot > \cdot \#b \ .^* < g \ c_m c_{m-1} \dots c_1 \bot > k \ .$$

That is, with an initial guess of the first variable to consider on top of the stack, is there a trace which pushes all clause labels on top of the stack?

First we show that if ϕ is satisfiable, then there exists a valid trace satisfying our query: for every variable x_i that evaluates to false, we set the link x_i as failed, meaning that link $\overline{x_i}$ is available (and thus if x_i evaluates to true, the link x_i is available). Since ϕ is satisfied, for each clause C_i there is a variable $x_{v(C_i)}$ whose valuation satisfies the clause. Now starting from link b, we either use link $x_{v(C_1)}$ or $\overline{x_{v(C_1)}}$ depending on the valuation of $x_{v(C_1)}$ that satisfies the clause C_1. By construction of τ we can push c_1 and g onto the stack, and when subsequently returning to b and replacing g with $g_{v(C_2)}$. This process is repeated until all clause symbols are on the stack in order, thus giving a valid trace for the query. The example formula $\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})$ from Table III is satisfied by the valuation $(x_1, x_2, x_3) = (\mathit{true}, \mathit{false}, \mathit{false})$, and allows for the following trace satisfying the query:

$$(b, g_1\bot) \to (x_1, gc_1\bot) \to (b, g_3c_1\bot) \to (\overline{x_3}, gc_2c_1\bot) \ .$$

Next we show that given a trace satisfying the query, there exists a valuation satisfying ϕ. Observe that by construction and by the priority of traffic engineering groups, for every i there is at most one of links x_i or $\overline{x_i}$ that is used. Hence, we construct the valuation of the variables as follows: if $\overline{x_i}$ is used then the variable x_i is set to false, otherwise it is set to true. Now, by construction of the forwarding rules in τ, if c_j appears on the stack, then there is a valuation of a variable that satisfies C_j. Since the query asks for every clause label to appear on the stack, all clauses are satisfied by the above valuation. The example network from Table III has e.g. the following trace that satisfies the query with no failed links:

$$(b, g_2\bot) \to (x_2, gc_1\bot) \to (b, g_2c_1\bot) \to (x_2, gc_2c_1\bot) \ .$$

This trace represents the valuation $(x_1, x_2, x_3) = (\mathit{true}, \mathit{true}, \mathit{true})$ that satisfies the formula ϕ. We note that we used the variable x_2 to satisfy both of the clauses, and hence the valuation of the remaining two variables can be arbitrary (in our constructions we decided to assign them the value true). □

We remark that our NP-hardness proof relies on only one router with a number of self-loop links. The construction can be in a straightforward way modified (unfolded by creating m copies of the router in the network) so that the network is without self-loops and is even acyclic.

IV. From Networks to Automata

We shall now explain how to reduce the query satisfiability problem in a network with at most k failed links into a reachability problem in pushdown automata. We need to first introduce some standard definitions from formal languages.

A. Preliminaries

A *nondeterministic finite automaton* (NFA) is a 5-tuple $A = (S, \Sigma, \delta, s_0, s_f)$ where S is a finite set of states, Σ is a finite input alphabet, $\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$ is the transition function, $s_0 \in S$ is the initial state, and $s_f \in S$ is the accepting state. A *configuration* of an NFA is a pair $(s, w) \in S \times \Sigma^*$ of a state and a string over Σ. Let $C(A)$ be the set of all such configurations. We define the *transition relation* (using infix notation) $\to_\delta \subseteq C(A) \times C(A)$ by $(s, w) \to_\delta (s', w)$ if $s' \in \delta(s, \epsilon)$, and $(s, aw) \to_\delta (s', w)$ if $s' \in \delta(s, a)$ for any $w \in \Sigma^*$ and $a \in \Sigma$. By $\to_\delta^*$ we denote the transitive and reflexive closure of $\to_\delta$. A string $w \in \Sigma^*$ is *accepted* by A if $(s_0, w) \to_\delta^* (s_f, \epsilon)$. We denote the set of all accepted strings by $Lang(A)$. If the NFA is clear from the context, we use the notation $s \overset{a}{\Rightarrow} s'$ as a shorthand for $(s, aw) \to_\delta^* (s', w)$, where $a \in \Sigma$. Intuitively, $s \overset{a}{\Rightarrow} s'$ means that from s we can perform zero or more ϵ-transitions, followed by a and followed again by zero or more ϵ-transitions before we reach the state s'.

A *pushdown automaton* (PDA) is a 5-tuple $P = (Q, \Gamma, \lambda, q_0, q_f)$ where Q is a finite set of states, Γ is a finite stack alphabet, $\lambda : Q \times \Gamma \to 2^{Q \times \Gamma^*}$ is the transition function where we require that the co-domain is finite, $q_0 \in Q$ is the initial state, and $q_f \in Q$ is the final state. A *configuration* of a PDA is the pair $(q, h) \in Q \times \Gamma^*$ where q is the control state and h a sequence of stack symbols with the top of the stack being the left-most symbol. Let $C(P)$ denote the set of all configurations. The *transition relation* $\to_\lambda \subseteq C(P) \times C(P)$ between configurations is defined by $(q, \ell h) \to_\lambda (q', \alpha h)$ whenever $(q', \alpha) \in \lambda(q, \ell)$ and where $\ell \in \Gamma$ and $h \in \Gamma^*$. The transitive and reflexive closure of $\to_\lambda$ is denoted by $\to_\lambda^*$. We shall use the stack symbol $\bot$ to represent the bottom-of-the-stack in our constructions.

Our work relies on the fact that reachability in pushdown automata is decidable in polynomial time.

Theorem 2 ([37, 38]). Let $P = (Q, \Gamma, \lambda, q_0, q_f)$ be a pushdown automaton and let (q_0, h_0) and (q, h) be two of its configurations. It is decidable in polynomial time wheather (q, h) is reachable from (q_0, h_0), i.e. $(q_0, h_0) \to_\lambda^* (q, h)$.

B. Useful Automata Constructions

In our query language, we use regular expressions that allow the user to define the restrictions on the desirable packet routing through the network. In our algorithmic solution to this problem, we shall use the standard fact that regular expressions are equivalent with NFA (they generate the same class of regular languages).

Theorem 3. [39] Given a regular expression $a \in Reg(\Sigma)$ we can construct in linear time an equivalent NFA $A = (S, \Sigma, \delta, s_0, s_f)$ such that $Lang(A) = Lang(a)$.

1) Destructing PDA: We can now describe a simple method of simulating the computation of an NFA by a PDA such that the string to be read by the NFA is initially on the stack of the PDA that accepts (with an empty stack) if and only if the NFA accepts the given string.

Given an NFA $A = (S, \Sigma, \delta, s_0, s_f)$, we define the *destructing* PDA $P_d = (Q, \Gamma, \lambda, q_0, q_f)$ such that

- $Q = S$, $q_0 = s_0$, $q_f = s_f$,
- $\Gamma = \Sigma \uplus \{\bot\}$, and
- $\lambda(s, a) = \{(s', \epsilon) \mid s \overset{a}{\Rightarrow} s'\}$ for all $s \in Q$ and $a \in \Gamma$.

It is easy to observe that the destructing pushdown has the following property.

Theorem 4. Given an NFA $A = (S, \Sigma, \delta, s_0, s_f)$, the constructed PDA $P_d = (Q, \Gamma, \lambda, q_0, q_f)$ has a computation $(q_0, h\bot) \rightarrow_\lambda^* (q_f, \bot)$ if and only if $h \in Lang(A)$.

Proof. From the definition of the language accepted by an NFA, we know that $h = a_1 a_2 \ldots a_n \in Lang(A)$ if and only if $(s_0, h) \rightarrow_\delta^* (s_f, \epsilon)$, i.e. there exists a sequence $s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} s_2 \overset{a_3}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_f$. By the construction of the pushdown automaton, this is the case if and only if $(q_0, a_1 a_2 \ldots a_n \bot) \rightarrow_\lambda (s_1, a_2 \ldots a_n \bot) \rightarrow_\lambda \ldots \rightarrow_\lambda (q_f, \epsilon)$ as claimed by the theorem. □

2) Constructing PDA: We are now interested in building the constructing PDA that allows us to push on its stack any string (though in the reverse order as the top of the stack is on the left) that is accepted by a given NFA. We can use an analogous construction as in the destructing PDA but pushing the symbols instead of popping.

Let $w = w_0 w_1 \ldots w_n$ be a string. The reverse of w is defined as $w^R = w_n w_{n-1} \ldots w_0$. The reverse of a language L is given by $L^R = \{w^R \mid w \in L\}$. In our constructions, we shall use the following fact.

Theorem 5. [40] Given an NFA A, we can in linear time construct an NFA A^R recognizing the reverse language $Lang^R(A)$.

Let $A^R = (S, \Sigma, \delta, s_0, s_f)$ be a reversed NFA, then the *constructing* PDA $P_c = (Q, \Gamma, \lambda, q_0, q_f)$ is defined by

- $Q = S \uplus \{q_0, q_f\}$,
- $\Gamma = \Sigma \uplus \{\diamond, \bot\}$, and
- λ is defined by $\lambda(q_0, \bot) = \{(s_0, \diamond\bot)\}$ and $\lambda(s, \diamond) = \{(s', \diamond a) \mid s \overset{a}{\Rightarrow} s'\} \cup \{(q_f, \epsilon) \mid s = s_f\}$ and in all other cases $\lambda(s, a) = \emptyset$.

Now we can formulate the expected theorem.

Theorem 6. Given an NFA A, the constructed PDA $P_c = (Q, \Gamma, \lambda, q_0, q_f)$ has a computation $(q_0, \bot) \rightarrow_\lambda^* (q_f, h\bot)$ if and only if $h \in Lang(A)$.

Proof. Let $h = a_1 a_2 \ldots a_n \in Lang(A)$ which is the case if and only if in the reversed NFA $A^R = (S, \Sigma, \delta, s_0, s_f)$ there exists a sequence $s_0 \overset{a_n}{\Rightarrow} s_1 \overset{a_{n-1}}{\Rightarrow} s_2 \overset{a_{n-2}}{\Rightarrow} \ldots \overset{a_1}{\Rightarrow} s_f$. Let us consider the following computation in the constructed PDA P_c: $(q_0, \bot) \rightarrow_\lambda (s_0, \diamond\bot) \rightarrow_\lambda (s_1, \diamond a_n \bot) \rightarrow_\lambda (s_2, \diamond a_{n-1} a_n \bot) \rightarrow_\lambda \ldots \rightarrow_\lambda (s_f, \diamond a_1 a_2 \ldots a_n \bot) \rightarrow_\lambda (q_f, a_1 a_2 \ldots a_n \bot)$. This shows that $(q_0, \bot) \rightarrow_\lambda^* (q_f, h\bot)$ and establishes the implication from right to left. For the opposite direction, if $(q_0, \bot) \rightarrow_\lambda^* (q_f, h\bot)$ then clearly the computation in the PDA must start with $(q_0, \bot) \rightarrow_\lambda (s_0, \diamond\bot)$ and finish with $(s_f, \diamond h\bot) \rightarrow_\lambda (q_f, h\bot)$ for some h, which by the construction of the PDA implies that $h \in Lang(A)$ as required. □

C. Encoding of MPLS Networks as PDA

We can now define the last ingredient that we need for solving the query satisfaction problem. We shall describe how a packet routing in an MPLS model with at most k link failures can be simulated by a PDA. Instead of enumerating by brute-force all possible combination of k failed links, we define an *over-approximation* PDA that includes all valid MPLS packet routings (possibly with other superfluous PDA executions), and an *under-approximation* PDA where every computation in such a PDA has a corresponding packet routing in MPLS network (but not necessarily every valid MPLS trace is represented in this PDA).

1) Over-approximation: We shall now define the over-approximating PDA for up to k failed links that instead for remembering the current set of failed links (there are exponentially many such combinations in general) essentially assumes that at every router up to k of its outgoing links may fail. This is an over-approximation as the PDA may include traces that have in total more than k failed links.

Assume an MPLS network $N = (V, I, E, L, \tau)$ with maximum k link failures. By $\overline{Ops}$ we denote the set of all MPLS operation sequences and all its suffixes that appear in the routing table τ. We recall that the routing function τ maps a link and a label to a sequence of traffic engineering groups $\tau(e, \ell) = O_1 O_2 \ldots O_n$ where $O_c = \{(e_c^1, \omega_c^1), (e_c^2, \omega_c^2), \ldots, (e_c^{l_c}, \omega_c^{l_c})\}$ for all $1 \leq c \leq n$. The k-failure aware routing function $\tau^k(e, \ell)$ includes all possible routing outcomes under the assumption that in the router v that has the incoming link e up to k links outgoing from v fail. Formally, $\tau^k(e, \ell) = \bigcup_{j=1}^{i} O_j$ where i is the smallest index such that the cardinality of the set $\{e_1^1, e_1^2, \ldots, e_1^{l_1}, e_2^1, e_2^2, \ldots, e_2^{l_2}, \ldots, e_i^1, e_i^2, \ldots, e_i^{l_i}\}$ is larger than k.

We shall now present the definition of over-approximating PDA that for a given network and a path-restricting regular expression (represented by an NFA) constructs a PDA that only allows the execution of traces that are possible in the network and are at the same time accepted by the path-restricting NFA.

Definition 8 (Over-approximating PDA)**.** Given a network $N = (V, I, E, L, \tau)$, an NFA $A = (S, E, \delta, s_0, s_f)$, and link-failure constant k, we define a path-restricted pushdown automaton $P(N, A) = (Q, \Gamma, \lambda^+, q_0, q_f)$ where

- $Q = (S \times E \times \overline{Ops}\}) \uplus \{q_0, q_f\}$,
- $\Gamma = L \uplus \{\bot\}$, and
- we define λ^+ as follows:

$$\begin{aligned}
\lambda^+(q_0,\ell) =& \{((s,e,\epsilon),\ell) \mid s_0 \overset{e}{\Rightarrow} s\} && (1)\\
\lambda^+((s,e,\epsilon),\ell) =& \{(q_f,\ell) \mid s = s_f\} \cup && (2)\\
& \{((s',e',\omega),\ell) \mid s \overset{e'}{\Rightarrow} s', (e',\omega) \in \tau^k(e,\ell)\}\\
\lambda^+((s,e,\alpha\omega),\ell) =& \left\{ \begin{array}{c} ((s,e,\omega),\epsilon) \text{ if } \alpha = \mathit{pop} \\ ((s,e,\omega),\ell') \text{ if } \alpha = \mathit{swap}(\ell') \\ ((s,e,\omega),\ell'\ell) \text{ if } \alpha = \mathit{push}(\ell') \end{array} \right\} && (3)\\
\lambda^+(q_f,\ell) =& \emptyset && (4)
\end{aligned}$$

The construction mimics the behaviour of the NFA in the control states of the pushdown automaton (the first component in the tripple) and it represents the header as a sequence of labels stored on the pushdown stack. Rule 1 initializes the pushdown control state to one of the possible NFA states reachable by reading the current edge e that is stored in the second component of the control state. Rule 2 performs the next hop in the network, and it is applicable only if the third component in the control state (storing a sequence of MPLS operations to be performed) is empty. Depending on the routing table, the control state is updated with the NFA state that we can reach by fowarding the packet along the edge e' and the third component now remembers the sequence of MPLS operations to be (one-by-one) executed on the stack. Finally, rule 3 removes MPLS operations from the third component of the pushdown control state and applies them on the labels that are stored on the stack. Once the final state q_f is reached (possible only if the NFA state in the first component is accepting—see rule2) then there are no further pushdown rules defined. We can now state a key property of our encoding.

Theorem 7. Let $N = (V, I, E, L, \tau)$ be a network, $A = (S, E, \delta, s_0, s_f)$ an NFA, and k a link-failure constant. For any trace $(e_0, h_0), (e_1, h_1), \ldots, (e_n, h_n)$ in the network with the set of failed links F such that $|F| \leq k$ and $e_0 e_1 \ldots e_n \in \mathit{Lang}(A)$, it holds that $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} (q_f, h_n\bot)$ in the constructed over-approximating PDA $P(N, A) = (Q, \Gamma, \lambda, q_0, q_f)$.

Proof. Let $(e_0, h_0), (e_1, h_1), \ldots, (e_n, h_n)$ be a trace in the network with failed links F such that $|F| \leq k$ and $e_0 \ldots e_n \in \mathit{Lang}(A)$. The proof is by induction over the length of this trace, with the induction hypothesis that for the prefix $(e_0, h_0), \ldots, (e_i, h_i)$ it holds that $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} ((s, e_i, \epsilon), h_i\bot)$ for every $s \in S$ s.t. $s_0 \overset{e_0 e_1 \ldots e_i}{\Longrightarrow} s$.

For the base case (where $i = 0$) the claim holds due to the PDA rule (1) for $\ell = \mathit{head}(h_0)$ that yields the PDA computation $(q_0, h_0\bot) \rightarrow_{\lambda^+} ((s, e_0, \epsilon), h_0\bot)$ for all s such that $s_0 \overset{e_0}{\Rightarrow} s$.

Now assume that $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} ((s, e_i, \epsilon), h_i\bot)$ is a PDA computation corresponding to the prefix $(e_0, h_0), \ldots, (e_i, h_i)$ where $i > 0$. To prove our claim, we want to show that $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} ((s, e_{i+1}, \epsilon), h_{i+1}\bot)$. By Definition 4, the existence of the transition from (e_i, h_i) to (e_{i+1}, h_{i+1}) in the network trace implies that $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, \mathit{head}(h_i)))$, implying that $(e_{i+1}, \omega) \in \tau^k(e_i, \mathit{head}(h_i))$. This means that by rule (2) we have $((s, e_i, \epsilon), h_i\bot) \rightarrow_{\lambda^+} ((s', e_{i+1}, \omega), h_i\bot)$ for every s' such that $s \overset{e_{i+1}}{\Longrightarrow} s'$. Since rule (3) simulates $\mathcal{H}$, we have $((s', e_{i+1}, \omega), h_i\bot) \rightarrow^*_{\lambda^+} ((s, e_{i+1}, \epsilon), h_{i+1}\bot)$, and thus $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} ((s', e_{i+1}, \epsilon), h_{i+1}\bot)$ for all s' such that $s_0 \overset{e_0 e_1 \ldots e_{i+1}}{\Longrightarrow} s'$.

By the fact that $e_0 e_1 \ldots e_n \in \mathit{Lang}(A)$ we can now conclude that $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} ((s_f, e_n, \epsilon), h_n\bot)$ and one final application of rule (2) yields $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} (q_f, h_n\bot)$. □

Remark 4. Notice that if there are no link failures (in other words if $k = 0$) then Theorem 7 holds also in the other direction, i.e. for any trace $(q_0, h_0\bot) \rightarrow^*_{\lambda^+} (q_f, h_n\bot)$ in $P(N, A) = (Q, \Gamma, \lambda, q_0, q_f)$ there exist edges $e_1, \ldots, e_n$ such that $e_0 \ldots e_n \in \mathit{Lang}(A)$ and $(e_0, h_0), \ldots, (e_n, h_n)$ is a valid trace in $N = (V, I, E, L, \tau)$. This easily follows from the fact that the routing function τ^0 never routes over backup links and hence the analysis is exact.

2) Under-approximation: The over-approximating pushdown we constructed above allows us to consider up to k failed links at every router, however, in the actual network we consider a fixed set F of failed links that does not change during the trace. Hence our over-approximation allows us to select some traffic engineering groups with lower priorities than those that can be possibly applicable for the fixed set of failed links. As a result, if there is routing (trace) in the MPLS network then there is a corresponding computation in the over-approximating pushdown. However, the existence of a PDA computation does not necessarily imply the feasibility of the corresponding routing in the network. For this reason, we suggest now also an under-approximating pushdown construction that excludes all superfluous pushdown traces. The intuition is to reuse the over-approximating pushdown construction where we add a fourth component to the control state (representing a counter of encountered failed links during the routing), so that the control states have the form (s, e, ω, i) where $0 \leq i \leq k$ such that i is the number of failed links so far. We thus define under-approximating PDA transition function λ^- as follows.

$$\begin{aligned}
\lambda^-(q_0,\ell) =& \{((s,e,\epsilon,0),\ell) \mid s_0 \overset{e}{\Rightarrow} s\}\\
\lambda^-((s,e,\epsilon,i),\ell) =& \{((s',e',\omega,j),\ell) \mid\\
& \quad ((s',e',\omega),\ell) \in \lambda^+((s,e,\epsilon),\ell)\\
& \quad \text{such that } j = i + h \leq k\\
& \quad \text{where } h \text{ is the smallest index with}\\
& \quad (e',\omega) \in \tau^h(e,\ell)\}\\
\lambda^-((s,e,\omega,i),\ell) =& \{((s,e,\omega',i),\alpha) \mid \omega \neq \epsilon\\
& \quad \text{and } ((s,e,\omega'),\alpha) \in \lambda^+((s,e,\omega),\ell)\}\\
\lambda^-(q_f,\ell) =& \emptyset
\end{aligned}$$

These rules simply add the number of failed links needed to activate a given traffic engineering group into the global counter, making sure that this total value does not exceed the maximum allowed number of failed links k. The problem with this construction is that if during the trace the same server is visited more than once, we can actually forward a packet along a link that we claimed as failed during the first visit of the router. However, a repeated router in a trace can be easily detected and our under-approximation becomes inconclusive if such a loop exists. We say that a computation in the under-approximating pushdown has a *loop* if the computation contains two different control states of the form (s, e, ϵ, j) and (s', e', ϵ, j') such that the links e and e' share the same source router.

Theorem 8. Let $N = (V, I, E, L, \tau)$ be a network, $A = (S, E, \delta, s_0, s_f)$ and NFA, and k a link-failure constant. If $(q_0, h_0\bot) \rightarrow^*_{\lambda^-} (q_f, h_n\bot)$ is a computation without a loop in the under-approximating pushdown $P(N, A) = (Q, \Gamma, \lambda, q_0, q_f)$ then there is a valid trace $(e_0, h_0), (e_1, h_1), \ldots, (e_n, h_n)$ in the network for some set of failed links F such that $|F| \leq k$ and $e_0 \ldots e_n \in \mathit{Lang}(A)$.

Proof. First, we notice that if we in the PDA computation $(q_0, h_0\bot) \rightarrow^*_{\lambda^-} (q_f, h_n\bot)$ consider only configurations with the control states of the form (s, e, ϵ, j) where the third component is ϵ, these naturally define a corresponding network trace $(e_0, h_0), (e_1, h_1), \ldots, (e_n, h_n)$. Moreover, every time the number j increases, we select an arbitrary set of failed links that activate $(e', \omega) \in \tau^h(e, \ell)$. We combine all these sets of failed links along the PDA execution into the set F. Clearly, $|F| \leq k$ and since the trace is loop free, we can pick these sets independently. Finally, by the construction of the under-approximating PDA, the network trace is valid and $e_0 \ldots e_n \in \mathit{Lang}(A)$. □

D. Solving the Query Satisfiability Problem

We have now described all the necessary automata constructions and are ready to provide a solution to the problem of query satisfiability in a given MPLS network model. Assume a given MPLS network model N and a query $< a > b < c > k$. We shall describe a construction of a final pushdown automaton P_{final} together with a reachability question that answers the query satisfiability problem. First, we construct (either using over- or under-approximation) the pushdown $P(N, A)$ where A is the NFA for the path-restricting regular expression b. Recall that $P(N, A)$ has the property stated in Theorem 7 resp. Theorem 8.

For the regular expression a representing the initial header, we create a constructing PDA P_c that satisfies the property in Theorem 6 and for the regular expression c representing the final header, we construct the destructing PDA P_d with the property in Theorem 4. Finally, we combine $P(N, A)$, P_c and P_d into a single pushdown automaton by running first P_c and once it enters its accepting state, we continue with the execution of $P(N, A)$ and once it accepts we run P_d as the last pushdown in this sequential composition. Let us call the resulting pushdown P_{final}. Building on the theorems presented earlier in this section, we can now conclude with the main result of our paper.

Theorem 9. Let $N = (V, I, E, L, \tau)$ be an MPLS network and let $q = < a > b < c > k$ be a query on N.

- Let $P_{\mathit{final}} = (Q, \Gamma, \lambda, q_0, q_f)$ be a pushdown automaton constructed above using the over-approximation. If there is a trace in the network N that satisfies the query q for a set of failed links F such that $|F| \leq k$ then $(q_0, \bot) \rightarrow^*_\lambda (q^f, \bot)$ in the pushdown automaton P_{final}.
- Let $P_{\mathit{final}} = (Q, \Gamma, \lambda, q_0, q_f)$ be a pushdown automaton constructed above using the under-approximation. If $(q_0, \bot) \rightarrow^*_\lambda (q^f, \bot)$ is a pushdown computation without a loop then there is a trace in the network N that satisfies the query q for some set of failed links F where $|F| \leq k$.

Proof. For the first claim, assume that there is a trace in the network N that satisfies the query q with at most k failed links. The initial header in the trace can be constructed in the pushdown automaton from the initial configuration $(q_0, \bot)$ due to Theorem 6, then Theorem 7 guarantees that we can execute the trace as a pushdown computation and finally Theorem 4 implies that we can pop the final header in the trace and reach the pushdown configuration $(q^f, \bot)$ as requested.

For the second claim, by Theorem 6 we know that the initial header must be the one satisfying the regular expression in the query q. Then Theorem 8 implies that the pushdown computation corresponds to a real network trace, and finally because we were able to pop the final header and reach the configuration $(q^f, \bot)$, we know by Theorem 4 that the header satisfies the requirement in the query q. □

We can see that the problem of query satisfiability in an MPLS network is in polynomial time reduced to a reachability problem in the automaton P_{final}, and thanks to Theorem 2, this problem can be solved in polynomial time. A negative answer to the reachability problem in the over-approximating pushdown automaton P_{final} implies that the given query is not satisfied in the network. A positive and loop-free answer to reachability in the under-approximating pushdown automaton implies that the given query is satisfied in the network. Otherwise the answer to the query satisfiability problem is inconclusive. In our implementation, we moreover run a (fast) check whether a trace that is returned by an over-approximation or by under-approximation (in case it has a loop), is executable in the MPLS network. If this is the case, the query is satisfied and the valid trace is output.

E. Removal of Redundant PDA Rules

A typical network contains a large number of routing table entries with many different labels. The reduction to PDA has to consider the possibility that each labels can be relevant at every router and create a pushdown rule for this case. However, several of such rules may be redundant, meaning that in the real execution there is no sequence of transitions that enable them. To remedy this, we present a pushdown reduction technique (based on static analysis of the produced pushdown automaton) for removing a large portion of the redundant rules.

That is to say, for a given PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ we want to find a reduced transition function λ' where $\lambda'(q, \ell) \subseteq \lambda(q, \ell)$ for all $q \in Q$ and all $\ell \in \Gamma$ such that

$$(q_0, \bot) \longrightarrow^*_\lambda (q_f, \bot) \text{ if and only if } (q_0, \bot) \longrightarrow^*_{\lambda'} (q_f, \bot) \ .$$

We first compute the function `find_tops(P, ℓ0)` presented in Algorithm 1 that over-approximates the top of the stack symbols in all reachable configurations of the pushdown

```
1  Function find_tops($P$, $\ell_0$)
       Input : A PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and $\ell_0 \in \Gamma$.
       Result: Top-of-the-stack function $T : Q \to 2^\Gamma$.
2      $Rules \leftarrow \{(q, \ell, q', \alpha) \mid (q', \alpha) \in \lambda(q, \ell)\}$;
3      Initialize two functions $T, S : Q \to 2^\Gamma$.
4      for $q \in Q$ do
5          $T[q] \leftarrow \emptyset$;
6          $S[q] \leftarrow \emptyset$;
7      end
8      $T[q_0] \leftarrow \{\ell_0\}$;
9      repeat
10         for $(q, \ell, q', \alpha) \in Rules$ do
11             if $\ell \in T[q]$ then
12                 $S[q'] \leftarrow S[q'] \cup S[q]$;
13                 if $|\alpha| \geq 1$ then
14                     $T[q'] \leftarrow T[q'] \cup \{\mathrm{head}(\alpha)\}$;
15                     $S[q'] \leftarrow S[q'] \cup \mathrm{tail}(\alpha)$;
16                 else
17                     $T[q'] \leftarrow T[q'] \cup S[q]$;
18                 end
19             end
20         end
21     until functions $T$ and $S$ do not change;
22     return $T$;
23 end
```

Algorithm 1: Approximation of top of the stack symbols

P starting in the initial configuration (q_0, ℓ_0). Recall that the *head* function yields the leftmost symbol of a stack, and we define the *tail* of a stack as the *set* of symbols that appear under the *head* symbol. The algorithm returns a function $T : Q \to 2^\Gamma$ that satisfies the following lemma.

Lemma 1. Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and an initial label ℓ_0, the algorithm find_tops(P,ℓ_0) terminates and returns a function T such that whenever $(q_0, \ell_0) \to_\lambda^* (q, \gamma)$ then $\mathrm{head}(\gamma) \in T[q]$.

Proof. Algorithm 1 clearly terminates as the functions T and S are monotonically growing and there are finitely many stack symbols in the alphabet Γ. By induction on n we prove the following claim: if $(q_0, \ell_0) \to_\lambda^n (q, \gamma)$ then after the termination of Algorithm 1 it holds that $\mathrm{head}(\gamma) \in T[q]$ and $\mathrm{tail}(\gamma) \subseteq S[q]$.

The claim clearly holds for $n = 0$ as the label ℓ_0 is added to $T[q_0]$ for the initial state q_0 at line 8. Assume now that the claims holds for $n > 0$. Let $(q_0, \ell_0) \to_\lambda^n (q, \gamma) \to_\lambda (q', \gamma')$. By induction hypothesis we assume that $\mathrm{head}(\gamma) \in T[q]$ and $\mathrm{tail}(\gamma) \subseteq S[q]$ and want to show that $\mathrm{head}(\gamma') \in T[q']$ and $\mathrm{tail}(\gamma') \subseteq S[q']$. Let us assume that the transition $(q, \gamma) \to_\lambda (q', \gamma')$ is due to the application of the rule $(q, \ell, q', \alpha) \in Rules$ such that $\gamma = \ell\beta$ and $\gamma' = \alpha\beta$. Let us assume the first iteration of the main repeat-loop where (by the induction hypothesis) both of the functions satisfy $\mathrm{head}(\gamma) \in T[q]$ and $\mathrm{tail}(\gamma) \subseteq S[q]$. Clearly, in the next iteration we get $\ell \in T[q]$ and the assignment at line 12 guarantees that at the end of the iteration of the repeat-loop we have $S[q] \subseteq S[q']$ and moreover if $|\alpha| \geq 1$ then by line 15 also $\mathrm{tail}(\alpha) \subseteq S[q']$. This implies that $\mathrm{tail}(\gamma') \subseteq S[q']$ as required. Furthermore, if $|\alpha| \geq 1$ then by line 14 we get $\mathrm{head}(\alpha) = \mathrm{head}(\gamma') \in T[q']$ and if $\alpha = \epsilon$ then by line 17 also $S[q] \subseteq T[q']$ which means that $\mathrm{head}(\gamma') = \mathrm{head}(\beta) \in T[q']$. The claim is thus established. □

We are now ready, for a given PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and the bottom of the stack symbol $\bot \in \Gamma$, to construct the reduced PDA $P' = (Q, \Gamma, \lambda', q_0, q_f)$ as follows.

1) Define the transition function $\mathit{reduce}(\lambda)$ by

$$\mathit{reduce}(\lambda)(q, \ell) = \{\ (q', \alpha) \in \lambda(q, \ell) \mid$$
$$q' = q_f \quad \text{or} \quad \alpha = \epsilon \quad \text{or} \quad \lambda(q', \mathrm{head}(\alpha)) \neq \emptyset\ \}$$

that removes rules from λ and we apply the function *reduce* repeatedly until no more rules can be removed. We denote the resulting transition function $\mathit{reduce}^*(\lambda)$.

2) Let T be the output of Algorithm 1 when called on the pushdown automaton P with the transition function $\mathit{reduce}^*(\lambda)$ and the stack symbol $\ell_0 = \bot$. We define the final reduced transition function λ' by

$$\lambda'(q, \ell) = \{\ (q', \alpha) \in \mathit{reduce}^*(\lambda)(q, \ell) \mid \ell \in T[q]\ \}\ .$$

Theorem 10. The reduced transition function λ' satisfies that $(q_0, \bot) \longrightarrow_\lambda^* (q_f, \bot)$ if and only if $(q_0, \bot) \longrightarrow_{\lambda'}^* (q_f, \bot)$.

Proof. The implication from right to left is trivial, as every rule in λ' is also a rule in λ. The implication from left to right follows from the fact that a single application of the function *reduce* removes only rules (q, ℓ, q', α) where $|\alpha| \geq 1$ such that there is no rule in control state q' under the action $\mathrm{head}(\alpha)$. Clearly removing (q, ℓ, q', α) does not change the possibility of reaching the control state q_f. This holds also for repeated application of the function *reduce*. Finally, due to Lemma 1, it is save to remove all rules that are not enabled in any reachable configuration, i.e. the stack symbol ℓ does not belong to $T[q]$. □

V. Implementation and Evaluation

Since our initial publication of P-Rex in the conference paper [31], we have fully re-implemented the method in C++, using the Moped engine [27] as a backend verifier for pushdown systems. In what follows, we discuss our experiences and experiments with the tool, both for synthetic scenarios (in order to be able to compare P-Rex with other approaches) and we as well report on an industrial case study in collaboration with a network operator NORDUnet (http://www.nordu.net). The source code of our tool is available at https://github.com/DEIS-Tools/AalWiNes.git and it is released under GPLv3 and online demo of the tool is accessible at http://demo.aalwines.cs.aau.dk. The HSA comparison experiments are executed on AMD Opteron 6376 processor with all files residing on an NVMe disk to reduce measurement noise while the remainder of the experiments are executed on AMD

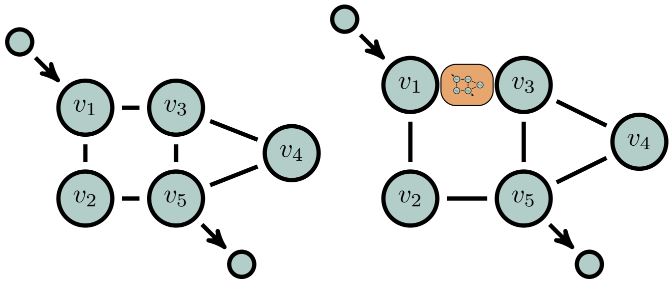

Fig. 3: Scaling of our synthetic network

P-REX HSA	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Nesting: 0 Routers: 5	0.044 0.015	0.048 0.038	0.046 0.143	0.044 0.259
Nesting: 1 Routers: 10	0.052 0.060	0.053 0.266	0.053 0.458	0.052 0.858
Nesting: 2 Routers: 15	0.063 0.054	0.062 0.247	0.060 0.976	0.062 10.160
Nesting: 3 Routers: 20	0.069 0.064	0.068 0.161	0.071 3.335	0.076 46.203
Nesting: 4 Routers: 25	0.080 0.050	0.081 0.221	0.080 5.532	0.080 146.323
Nesting: 5 Routers: 30	0.087 0.079	0.087 0.327	0.088 19.758	0.088 385.632
Nesting: 6 Routers: 35	0.096 0.028	0.096 0.452	0.094 18.725	0.097 731.246

TABLE IV: P-REX vs HSA runtime (in seconds)

P-REX HSA	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Nesting: 0 Routers: 5	0.046 0.025	0.045 0.055	0.046 0.152	0.046 0.220
Nesting: 0 Routers: 10	0.060 0.015	0.055 0.040	0.054 0.151	0.055 0.616
Nesting: 0 Routers: 15	0.060 0.014	0.064 0.075	0.064 0.742	0.063 8.323
Nesting: 0 Routers: 20	0.068 0.015	0.072 0.107	0.071 1.794	0.071 34.004
Nesting: 0 Routers: 25	0.076 0.016	0.081 0.133	0.077 3.432	0.081 100.408
Nesting: 0 Routers: 30	0.084 0.016	0.087 0.173	0.087 5.638	0.089 231.513
Nesting: 0 Routers: 35	0.092 0.016	0.094 0.204	0.095 8.360	0.096 465.846

TABLE V: P-REX vs HSA runtime (in seconds) without creating any label nesting

EPYC 7551 processors with all files residing on a SAN. All experiments were limited to 16 GB of memory usage using release `v0.92.J` of our tool. Furthermore, we also provide a repeatability-package [41].

A. Comparing HSA and P-REX

We compare the performance of P-REX to HSA [18], in a series of scalable instances by considering the simple network from Figure 3 on the left. This synthetic topology contains two shortest paths between v_1 and v_5 and one backup path via v_4 in case of a link failure. It is chosen to be a minimal example on which we can show the effect of nesting the network topology recursively. The scaling is achieved by repeatedly replacing the link between v_1 and v_3 by the same subnetwork (as illustrated in Figure 3 on the right). This increases both the number of routers as well as the nesting of labels in MPLS headers, as each nesting creates an additional MPLS tunnel. In the other dimension, we scale the number of failed links in the network from 0 to up to 3 failed links. In the HSA tool, we encode nested MPLS labels as a product label (sequence of labels) but the number of such new product labels grows exponentially in the nesting depth. Regarding the failover protection, HSA simply enumerates all possible sets of failed links and performs a reachability analysis for each such configuration, whereas P-REX uses the over-approximation approach: the concrete query is

$$< \textbf{ip} > [\cdot\#v_1] \cdot^* [v_5\#\cdot] < \cdot\cdot >$$

which is not satisfied and hence our answer is conclusive. In Table IV we can see the runtime in seconds rounded up to three decimal digit for P-REX (top part of each entry) vs. HSA (bottom part of each entry). We can see that once we scale the number of failed links or the size of the network (including the nesting depth of MPLS labels), the performance of HSA deteriorates significantly. On the other hand, our tool conclusively answered the query for all instances in less than 0.1 second.

Next, in Table V we run the same experiment by still scaling the number of routers in the network, however, without increasing the nesting of labels (in order words, the header contains only one MPLS label at any moment). Note that even though the numbers in the first row of both Table IV and V should be the same, they differ slightly due to the experimential noise (only within the the range of a few milliseconds). The experiments in Table V show that removing the nesting of labels reduces in general the running time of HSA (sometimes even to one half) but the HSA approach still causes exponential explosion with the increasing size of the network as well as with the increasing number of failed links.

In conclusion, this experiment demonstrates that our tool scales significantly better both in the nesting depth of the MPLS labels and the size of the network as well as in the number of considered failed links. This is caused by the fact that we do not enumerate all sets of possibly failed links and that we treat the MPLS labels as independent stack symbols in our pushdown automaton, whereas other existing tools (including HSA) must encode the label-stack as a tuple of stack symbols and this causes a significant explosion in the number of possible packet headers (exponential in the number of tunnels).

B. Industrial Case Study

Next, we report on a case study performed on a real-world MPLS network operated by NORDUNET, a regional service provider with a network consisting of 24 MPLS routers, geographically distributed across several countries. The routers are primarily Juniper, running JunOS and their MPLS network employs more than 20,000 significant labels. The forwarding tables format is documented in [42].

P-REX consists of a fully automated toolchain. In addition to the forwarding tables, to obtain topological adjacency information, we also use the *Intermediate System to Intermediate Systems (IS-IS)* database. To extract the information from the routers, we run the following commands on each router in the network:

- `show isis adjacency detail | display xml`
- `show route forwarding-table family mpls extensive | display xml`
- `show pfe next-hop | display xml`

The obtained data is automatically parsed in order to construct a corresponding pushdown automaton. In total, the number of forwarding rules collected from NORDUNET network amounts to almost 600,000 pushdown rules in our model. Notice that while JunOS only matches on the incoming top label, our model matches on both the incoming interface as well as the incoming top label, and thus P-REX adds the forwarding entries for all interfaces, increasing the number of rules by a constant factor. From the XML-output from JunOS, P-REX can automatically construct the PDA and subsequently call MOPED and conduct trace-validation.

1) Reachability Matrix: In order to demonstrate the feasibility of our approach on the network provided by NORDUNET, we compute the reachability matrix between any pair of routers $R1$ and $R2$ (router names are anonymized and indexed by A to X) for increasing number of failed links. If a given reachability query between a pair of routers is not satisfied (this can be proved only using over-approximation), we denote it by a dot. If the query is satisfied, we use the symbol ✓ in case that the over-approximation returned a trace that was verified as a valid trace in the network. The over-approximation can however also return an invalid trace and in this case we run under-approximation and if it returns a trace, we denote it by the symbol $\underline{\checkmark}$. A question mark means that the query verification was inconclusive (over-approximation returned true but the returned trace was invalid and under-approximation returned false).

Tables VI, VII, VIII and IX study the connectivity of routers in the core network with an initial header consisting of a single MPLS label. We can see that a one link failure introduces several new connections in Table VII compared to Table VI (e.g. the connection between B and F). Only a single answer in Table VII is rendered inconclusive, namely a self-loop on B. For the remaining queries, we can see that 15 connections require the under-approximation technique in order to obtain a conclusive answer.

In Tables VIII and IX, we study the connectivity to routers participating in the transit path (with possibly additional MPLS labels) of the connections found in Table VI and Table VII (respectively). The tables are almost identical (in terms of connectivity) with a few additional connections. We also note that 20 queries need the under approximation to prove connectivity and two answers are inconclusive.

If we instead focus on IP-to-IP over MPLS connectivity, Table X shows a significantly sparser connectivity matrix (rows with no connections are left out). This behaviour is expected because direct (non-tunneled) IP-to-IP connectivity is excluded from the configurations received from NORDUNET and IP routing is in general not common within their core network. This implies that we only observe IP-to-IP connections if they are implemented via MPLS label switching.

When we study the transit-routers of the IP routing implemented over MPLS (Tables XI, XII and XIII), we can see that an increased number of failed links leads to more connectivity in the grid (e.g. link (A, G)). This highlights the internal use of internal MPLS tunneling for fail-over routes with IP traffic in the core network.

We also notice that the routers O, R and T never participate in communication in the MPLS-part of the network. A closer inspection of their configurations reveals that these routers only communicate with edge-routers (not shown in the table) via the so-called service MPLS labels, and thus are unreachable from the internal routers.

Finally, in Table XIV we present an overview showing how many positive and negative techniques the over and under-approximation techniques answered for the reachability matrices in Tables VI-IX. For the queries with $k = 0$, both over and under-approximations answered the same number of queries. This is because without any failed links both methods are exact. For the two tables with $k = 1$, we can see that under-approximation is in general slightly better in answering the positive queries but it is not able to falsify any of the queries as it can be only used to find counter-examples. In total, only in three instances the answer was inconclusive (neither over nor under-approximation provided an answer).

Regarding the verification time per query, in the current C++ re-implementation of the tool, we use on average 1.87 second to answer a query in the reachability matrix plus 3.09 seconds are used to parse the network and the query, whereas in the prototype tool implementation presented in our conference paper [31], it took on average 1 hour and 1 minute to parse the network and answer a single query. This is a result of an optimized implementation, improved reductions on pushdown automata and the use of C++ (earlier implementation was done in Python).

2) Operator Specific Queries: During our in-person meetings with NORDUNET, we identified the following relevant queries that served as a test case for our workload. The operator asked whether a packet arriving at A with the sticky header 449552 on top of an IP-address can be routed to B via F. The P-REX query is formulated as

$$< [\$449552]\,\mathbf{ip} > [.\#A] . * [.\#F] . * [.\#B] < \mathbf{ip} > 0$$

and in case of no failed links, our tool is able to conclude in 1.26 seconds while using 750 MB of memory that this is not the case. However, if the initial label is substituted with any of the labels $449534, $449535, $449546, $449547, $449548, $449549, $449550, $449568, $454741, $454742, $454754, $454758, $454761, $454774, $454776, $454780, $455253, $455283, the query becomes satisfied. In fact, we can check that this set of labels covers all the satisfiable instances of the query by checking

$$< [\hat{}\$449535, \ldots, \$455283]\,\mathbf{ip} > [.\#A] . * [.\#F] . * [.\#B] < \mathbf{ip} > 2$$

TABLE VI: Reachability matrix for
$< \textbf{smpls ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < \textbf{smpls ip} >\ 0$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	S	U	V	W	X
A	.	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	.
B	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
C	✓	.	.	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	.
D	✓	.	✓	✓	✓	✓	✓	.	.	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	.
E	✓	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	.
F	✓	.	✓	✓	✓	.	✓	✓	✓	✓	.	.	✓	.	✓	✓	.	.	✓	✓	.
G	✓	.	✓	✓	✓	✓	.	✓	✓	✓	.	.	✓	.	✓	✓	.	✓	.	✓	.
H	✓	.	✓	.	✓	✓	✓	.	✓	✓	.	.	.	.	✓	✓	.	.	.	✓	.
I	.	.	.	.	.	✓	✓	✓	.	✓	.	.	.	.	.	✓	.	.	.	✓	.
J	✓	.	✓	.	✓	✓	✓	✓	✓	.	.	.	✓	.	✓	✓	.	✓	.	✓	.
K	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
L	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
M	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	.	.	.	.	✓	✓	.	✓	✓	✓	.
P	✓	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	.	✓	.	✓	✓	✓	.
Q	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	.	.	✓	.	✓	.	.	✓	✓	✓	.
U	✓	.	✓	.	✓	✓	✓	.	✓	✓	.	.	✓	.	✓	✓	.	.	.	✓	.
V	✓	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	.	.	✓	.	✓	.
W	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	.	.	✓	.	✓	✓	.	✓	.	.	.
X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

TABLE VII: Reachability matrix for
$< \textbf{smpls ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < \textbf{smpls ip} >\ 1$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	S	U	V	W	X
A	✓̲	.	✓	✓	✓̲	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	.
B	.	?	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
C	✓	.	✓̲	✓̲	✓̲	✓̲	✓	✓	.	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
D	✓	.	✓̲	✓̲	✓	✓	✓	.	.	✓	.	.	✓	.	✓	✓	.	✓	✓	✓	.
E	✓̲	.	✓̲	✓̲	✓̲	✓	✓	✓	.	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
F	✓	✓	✓	✓	✓	✓̲	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
G	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
H	✓	.	✓	.	✓	✓	✓	.	✓	✓	.	.	.	.	✓	✓	.	.	.	✓	.
I	✓	.	.	.	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	✓	.	✓	.	✓	✓
J	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
K	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
L	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
M	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	.
P	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓
Q	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	.
U	✓	.	✓	✓	✓	✓	✓	.	✓	✓	✓	.	✓	.	✓	✓	.	.	.	✓	.
V	✓	.	✓	✓	✓	✓	✓	✓	.	✓	.	.	✓	.	✓	✓	.	✓	.	✓	.
W	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓̲	.	✓	.	✓	✓	.	✓	✓̲	✓	✓
X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

TABLE VIII: Reachability matrix for
$< \textbf{smpls ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < (\textbf{mpls} *\ \textbf{smpls})?\ \textbf{ip} >\ 0$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	S	U	V	W	X
A	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
B	.	.	.	.	.	.	.	✓	.	.	.	✓	.	.	.	.	.	.	.	.	.
C	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	✓	.	✓	✓	✓	✓	✓	.	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓
E	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
H	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I	.	✓	.	.	.	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓	.	✓	✓
J	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
K	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
L	.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
M	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓
P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓
Q	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓
U	✓	.	✓	✓	✓	✓	✓	.	✓	✓	✓	.	✓	.	✓	✓	✓	.	✓	✓	✓
V	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
W	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓
X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.	.	.

TABLE IX: Reachability matrix for
$< \textbf{smpls ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < (\textbf{mpls} *\ \textbf{smpls})?\ \textbf{ip} >\ 1$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	S	U	V	W	X
A	✓̲	✓	✓̲	✓	✓̲	✓̲	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
B	.	?	.	.	.	✓	.	✓̲	.	.	.	✓	.	.	.	.	.	.	.	.	.
C	✓̲	✓	✓	✓̲	✓̲	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	✓	.	✓̲	✓	✓	✓	✓	.	✓	✓	✓̲	.	✓	✓	✓	✓	✓	✓	✓	✓	✓
E	✓̲	✓	✓̲	✓̲	✓	✓	✓	✓	✓	✓	✓̲	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F	✓̲	✓	✓	✓	✓	✓̲	✓	✓̲	✓	✓	✓	✓	✓	✓	?	✓	✓	✓	✓	✓	✓
G	✓	✓	✓	✓	✓	✓	✓̲	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
H	✓	✓	✓	✓	✓	✓	✓	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I	✓	✓	.	.	✓	✓	✓	✓	.	✓	.	✓	✓	.	✓	✓	✓	✓	.	✓	✓
J	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
K	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
L	.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
M	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Q	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
U	✓	.	✓	✓	✓	✓	✓	.	✓	✓	✓	.	✓	.	✓	✓	✓	.	✓	✓	✓
V	✓̲	.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓̲	✓	✓
W	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓	.	.	.

Reachability Matrices for the following queries.

A: $< \textbf{ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < \textbf{ip} >\ 0$	B: $< \textbf{ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < (\textbf{mpls} *\ \textbf{smpls})?\ \textbf{ip} >\ 0$
C: $< \textbf{ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < (\textbf{mpls} *\ \textbf{smpls})?\ \textbf{ip} >\ 1$	D: $< \textbf{ip} > [\cdot\#R1]\ \cdot^*\ [\cdot\#R2] < (\textbf{mpls} *\ \textbf{smpls})?\ \textbf{ip} >\ 2$

TABLE X: Query A

	A	C	D	E	F	G	P	U	V	W
A	.	.	.	✓	.	.	✓	.	.	.
C	.	.	.	✓	.	.	.	.	.	.
E	✓	.	.	.	.	.	.	.	.	.
F	.	.	.	.	.	.	✓	.	.	.
G	.	.	.	✓	✓	.	.	.	.	.
P	.	.	.	.	✓	.	.	.	.	.
V	.	.	✓	.	.	.	.	.	.	.

TABLE XI: Query B

	A	C	D	E	F	G	P	U	V	W
A	.	.	.	✓	✓	.	✓	.	.	.
C	.	.	.	✓	.	.	.	.	.	.
E	✓	.	.	.	.	.	.	.	.	.
F	.	.	.	.	.	.	✓	.	.	.
G	.	.	.	✓	✓	.	.	.	.	.
P	.	.	.	.	✓	.	.	.	.	.
V	.	.	✓	.	.	.	.	.	.	.

TABLE XII: Query C

	A	C	D	E	F	G	P	U	V	W
A	✓̲	.	.	✓	✓	✓	✓	.	.	.
C	.	.	.	✓	.	.	.	.	.	.
E	✓	.	.	.	.	.	.	.	.	.
F	✓	.	.	.	?	.	✓	.	.	.
G	✓	.	.	✓	✓	?	✓	.	.	?
P	.	.	.	.	✓	✓	?	.	.	.
V	.	?	✓	.	.	.	.	.	.	.

TABLE XIII: Query D

	A	C	D	E	F	G	P	U	V	W
A	✓	.	.	✓	✓	✓	✓	.	.	.
C	.	.	✓	✓	.	.	.	.	.	.
E	✓	✓	.	.	.	.	.	.	.	.
F	✓	.	.	✓	✓	✓	✓	.	.	.
G	✓	.	.	✓	✓	✓	✓	✓	.	✓
P	.	.	.	.	✓	✓	✓	.	✓	.
V	.	✓	✓	.	.	.	✓	.	.	.

Method	Over		Under		
Answer	true	false	true	false	inconclusive
Table VI ($k = 0$)	184	392	184	392	0
Table VII ($k = 1$)	213	347	228	0	1
Table VIII ($k = 0$)	294	282	294	282	0
Table IX ($k = 1$)	285	269	304	0	2

TABLE XIV: Number of queries from Tables VI-IX answered by over and under-approximation, devided into true and false queries.

which is unsatisfied, also in the case of 2 link-failures (verified in 8.14 seconds using 750 MB of memory).

Another interesting question is the maximal number of hops traversed by a packet arriving in the network with a service-label. By verifying the query

$$< \textbf{smpls ip} >\ \cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot * < \cdot * >\ 0$$

we can conclude that it is impossible to have more than 8 such hops without any link failures (computed in 21.5 seconds using 1148 MB). However, we can see that 8 hops are possible with a single link failure (28.38 seconds and 1141 MB).

TABLE XV: Effect of reduction pr. query (on average) for computing the queries from Table IX, using either no reduction, simple Top-Of-Stack reduction (TOS) from [31] or Improved TOS (ITOS) introduced in Section IV-E

	No Reduction	TOS [31]	ITOS
# Pushdown rules	967286	463698	3723
Moped verification time	5.97	2.41	0.02
Translation + Moped time	8.00	7.32	2.36

As a last query, we investigate the maximal tunneling depth of the network, which we assume is correlated directly with the maximal stack height. The query

$$< \mathbf{smpls}?\ \mathbf{ip} > .^{*} < ..\ \mathbf{smpls\ ip} >\ 0$$

is false for the network (computed in 166.28 seconds using 8978 MB) which tells us that the stack never grows beyond 2 labels high. The outcome of this query does not change with an increased number of failed links.

However, the query

$$< \mathbf{smpls}?\ \mathbf{ip} > .^{*} < .\ \mathbf{smpls\ ip} >\ 0$$

is satisfied (computed in 145.16 seconds using 6044 MB) and the resulting trace shows that a packet arriving at router A with the empty MPLS header (i.e. just the IP label) can have two labels pushed after which it is forwarded to the P router.

We remark that in the previous implementation of P-Rex from [31], used between 28 and 109 minutes on answering similar queries while using between 5 and 14 GB of memory.

C. Pushdown Reductions

Finally, we present experimental evidence regarding the efficiency of our pushdown reduction given in Section IV-E. We study its effect across the queries used for computing Table IX using the under-approximation PDA.

We can observe in Table XV that our ITOS reduction significantly reduces the number of rules in the resulting PDA (average over all verified queries), resulting in less than a percent of the rules compared to the pushdown before the reduction is applied. The TOS reduction presented in our conference paper [31] is only capable of halving the number of rules on average. The running-time of Moped engine is proportionally faster, dropping from 5.97 second on average to 0.02 seconds on average. Observe also that the running-time of the entire tool-chain (translation, reduction and verification) is reduced by more than 70% using ITOS and by less than 9% using only TOS. If we focus on the worst-case running-time without reduction, we can see that a computation for the query

$$< \mathbf{ip} > [\cdot\#C]\ \cdot^{*}\ [\cdot\#T] < (\mathbf{mpls} *\ \mathbf{smpls})?\, \mathbf{ip} > 2$$

takes 54.5 seconds to construct and verify with no reduction while the ITOS method reduces the problem to a trivially false, leaving just the computation time for the construction and reduction of 5.2 seconds. The overhead of computing the pushdown reduction is hence less significant and the overall running time improved by more than 3 times on average.

VI. Conclusion

While there is a wide consensus in the network community that networks should become more automated, it is less clear how to achieve this efficiently. Our work shows that, for the case of MPLS networks, there exist techniques that allow for a fast, polynomial-time analysis, even accounting for an exponential number of possible failure configurations. In particular, we present a what-if analysis tool P-Rex that significantly outperforms existing tools, as demonstrated on a case study in collaboration with a network operator.

We understand our work as a first step towards an industrial employment of our method, and believe that this paper opens several interesting directions for future research. In particular, we plan to explore the use of the CEGAR (counter-example guided abstraction) approach to further improve the performance P-Rex, and to add quantitative attributes like bandwidth and delay, by using a weighted extension of our automata-theoretic technique. Another interesting direction for future research includes the synthesis [20, 21] of correct-by-design network configurations.

Acknowledgments. We would like to Magnus Bergroth, Markus Krogh, Henrik Thostrup Jensen, and Dennis Wallberg from NORDUnet for answering our questions about their MPLS deployment and for providing us with the case study. This research was supported by the Vienna Science and Technology Fund (WWTF) under grant number ICT19-045 and the DFF project QASNET.

References

[1] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Proc. ACM SIGCOMM*. ACM, 2016, pp. 328–341.

[2] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016, pp. 217–232.

[3] M. Menth, M. Duelli, R. Martin, and J. Milbrandt, "Resilience analysis of packet-witched communication networks," *IEEE/ACM transactions on Networking (ToN)*, vol. 17, no. 6, pp. 1950–1963, 2009.

[4] A. K. Atlas and A. Zinin, "Basic specification for ip fast-reroute: loop-free alternates," IETF RFC 5286, 2008.

[5] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "Ip fast rerouting for multi-link failures," in *Proc. IEEE INFOCOM*. ACM, 2014, pp. 2148–2156.

[6] RFC 8001, "Rsvp-te extensions for collecting shared risk link group (srlg)," https://datatracker.ietf.org/doc/rfc8001/, 2017.

[7] D. Xu, Y. Xiong, C. Qiao, and G. Li, "Failure protection in layered networks with shared risk link groups," *IEEE Network*, vol. 18, no. 3, pp. 36–41, 2004.

[8] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 750–764.

[9] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 201–219.

[10] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *SIGPLAN Not.*, vol. 49, no. 1, pp. 113–126, 2014.

[11] R. Birkner, T. Brodmann, P. Tsankov, L. Vanbever, and M. T. Vechev, "Metha: Network verifiers need to be correct too!" in *NSDI*, 2021, pp. 99–113.

[12] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2015, pp. 469–483.

[13] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. (SIGCOMM)'17. ACM, 2017, pp. 155–168.

[14] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. (SIGCOMM)'16. ACM, 2016, pp. 300–313.

[15] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020, pp. 953–967.

[16] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Scalable verification of border gateway protocol configurations with an smt solver," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 765–780.

[17] P. G. Jensen, M. Konggaard, D. Kristiansen, S. Schmid, B. C. Schrenk, and J. Srba, "Aalwines: A fast and quantitative what-if analysis tool for mpls networks," in *Proc. 16th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[18] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012, pp. 113–126.

[19] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating mpls networks," in *IEEE International Conference on Computer Communications (INFOCOM'18)*. IEEE, 2018, pp. 1–9.

[20] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018, pp. 579–594.

[21] ——, "Network-wide configuration synthesis," in *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 2017, pp. 261–281.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41 (4). ACM, 2011, pp. 290–301.

[23] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2013, pp. 15–27.

[24] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2014, pp. 87–99.

[25] K. G. Larsen, S. Schmid, and B. Xue, "Wnetkat: A weighted sdn programming and verification language," in *Proc. 20th International Conference on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl. Leibniz-Zentrum für Informatik, 2016, pp. 1–18.

[26] D. M. Kahn, "Undecidable problems for probabilistic network programming," in *MFCS'17*, vol. 83. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: LIPIcs-Leibniz International Proceedings in Informatics, 2017, pp. 1–16.

[27] S. Schwoon, "Model-checking pushdown systems," Ph.D. Thesis, Technische Universität München, Jun. 2002. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf

[28] Duluth News Tribune, "Human error to blame in minnesota 911 outage," in *https://www.ems1.com/911/articles/389343048-Officials-Human-error-to-blame-in-Minn-911-outage/*, 2018.

[29] R. Chirgwin, "Google routing blunder sent japan's internet dark on friday," in *https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/*, 2017.

[30] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *arXiv preprint arXiv:1710.11583*, 2017.

[31] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-rex: Fast verification of mpls networks with multiple link failures," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 217–227. [Online]. Available: https://doi.org/10.1145/3281411.3281432

[32] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "Fsr: Formal analysis and implementation toolkit for safe interdomain routing," *IEEE/ACM Transactions on Networking (ToN)*, vol. 20, no. 6, pp. 1814–1827, 2012.

[33] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *European Symposium on Programming*. Springer, 2016, pp. 282–309.

[34] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 750–764.

[35] M. Chiesa, A. Kamisinski, J. Rak, G. Retvari, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys and Tutorials (COMST)*, 2021.

[36] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. Thomson Course Technology, 2006.

[37] J. Büchi, "Regular canonical systems," *Arch. Math. Logik u. Grundlagenforschung*, vol. 6, pp. 91–111, 1964.

[38] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model checking pushdown systems," in *Proc. 12th International Conference on Computer Aided Verification (CAV)*, ser. LNCS, vol. 1855. Springer, 2000, pp. 232–247.

[39] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[40] J. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," *Mathematical Theory of Automata*, vol. 12, pp. 529–561, 1962.

[41] P. G. Jensen, H. T. Jensen, I. van Duijn, J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "Experiments for "Automata Theoretic Approach to Verification of MPLS Networks under Link Failures"," Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3888242

[42] Juniper, "Show route forwarding-table," Technical Documentation https://www.juniper.net/documentation/en_US/junos/topics/reference/command-summary/show-route-forwarding-table.html, 2018.