P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures

Jesper Stenbjerg Jensen
Aalborg University
Denmark

Troels Beck Krøgh
Aalborg University
Denmark

Jonas Sand Madsen
Aalborg University
Denmark

Stefan Schmid*
University of Vienna
Austria

Jiří Srba†
Aalborg University
Denmark

Marc Tom Thorgersen
Aalborg University
Denmark

ABSTRACT

Future communication networks are expected to be highly automated, disburdening human operators of their most complex tasks. However, while 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, 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 and comes with an expressive query language based on regular expressions. It takes into account the actual router tables, and is hence well-suited for debugging. We also report on an industrial case study and demonstrate that P-Rex supports rich queries, performing what-if analyses in less than 70 minutes in most cases, in a 24-router network with over 100,000 MPLS forwarding rules.

CCS CONCEPTS

• **Networks** → Network reliability; **Network algorithms**;

KEYWORDS

Network Verification, MPLS, Prefix Rewriting Systems

1 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 undesired network behaviors, which is easily overlooked, such as datacenter traffic leaking to the Internet in unintended ways [5]. More generally, unintended behavior after failures can harm the availability, security, and performance of a network [12]. The possibility of *multiple* link failures [2, 10, 21], e.g., due to shared risk link groups [22, 27], 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. [1, 17]. 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 [1], in the worst case, and some queries are even undecidable [16, 19]. What is more, existing tools do not provide much support for reasoning about network behavior *under failures*, a major concern of operators responsible for the availability of the network. The few notable exceptions that do support some kind of what-if analysis accounting for failures share the drawback that they are highly inefficient as they mainly 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. Only recently has there been some effort to remedy this issue by instead analyzing the control plane while considering multiple data planes at the same time [3, 14], however, these methods are not yet applicable to MPLS.

We are interested in the *fast* and automated verification (even under failures) of networks based on Multiprotocol Label Switching (MPLS). 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 accummulated 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.

*Also affiliated with University of Aalborg, Denmark, and TU Berlin, Germany.
†Also affiliated with FI MU in Brno, Czech Republic.

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12.
https://doi.org/10.1145/3281411.3281432

1.1 Our Contributions

We present P-Rex, a *what-if analysis* tool supporting the fast verification of MPLS-based communication networks, accounting for the possibility of failures. In particular, P-Rex allows to test a wide range of important network properties in *polynomial-time*, independently of the number of failures. The runtime of 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

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

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 routing traces through the 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 enables it to find bugs. The tool also allows 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). Toward this end, we present over-approximation and under-approximation techniques to further improve performance.

Our experiments demonstrate a convincing performance of P-Rex compared to existing tools, on different workloads. For this comparison, we modified the HSA tool [17] 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 contaning over 100,000 forwarding table entries in less than an hour.

1.2 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). This PDA is then an input for the backend tool *Moped* [24] that is used for reachability analysis. P-Rex encodes the network as a PDA which can be then queried. 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. The three PDAs are combined into a single PDA which we give to the PDA reachability tool Moped [24]. Moped then either provides a trace through the pushdown which witnesses the query, or says that no such witness exists.

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 is used to restrict 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 state of the PDA: the top of stack reduction technique greatly reduces the amount of transitions in the PDA.

	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	✓	✓ [1]	✓	✓	✓
Waypointing	✓	✓	✓	✓	χ
Language	Py., C	OCaml	Py., C	Py.	C++, Ruby

Table 1: Comparsion of related tools

2 RELATED WORK

Motivated by the complexity and frequent errors of manual network operations, much progress has been made over the last years towards more automated and formally verifable networks [1, 4, 13, 17, 18, 20, 28]. Another driver for studying formal methods in networking is SDN.

Typically, network verification tools are given some model or configuration of the control plane or the data plane, and some properties to check. Table 1 provides an overview and comparison of several selected tools: Some tools are specific to a certain protocol, such as BGP [26] or OpenFlow (OF), others are protocol agnostic (agn.) [17]. 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. Most existing tools do not support arbitrarily large header sizes, which however is required for MPLS verification.

For example, NetKAT [1] 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) [17] 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 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. VeriFlow [18] 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 [20] 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 algorithms to detect reachability, forwarding loops, and packet loss as invariants.

However, none of the tools mentions failures, and at best, requires the network operator to enumerate all failure scenario combinations, which comes at a high runtime cost. Another strength of P-Rex is that it operates directly on the actual routing information, and not on its logical abstraction, which allows to find bugs.

Our earlier work [23] provides the theoretical underpinnings upon which our tool builds. However, to apply this theory in practice, the underlying MPLS network model has to be generalized and a number of research challenges have to be solved. First of all, a query language is required which strikes a balance between being compact and yet intuitive to use. The expressiveness of the query language has been extended by adding regular expressions over path quantifiers and hence allowing us to ask about a routing that e.g., avoids certain routers, a query that was impossible to verify in [23]. Also the formal MPLS model has been significantly extended to account for non-determinism (to model traffic engineering) and multi-operation chains (motivated by the Juniper router configurations in our case study), and parallel links between routers. Our extensions also motivate us to introduce novel over- and under-approximation techniques to improve performance further. Performance is further improved through a more compact network model with fewer transitions. P-Rex introduces a novel method for combining the NFA generated from a query and the PDA into a single PDA which then simulates the two automata running in lockstep, as well as several performance optimizations necessary to verify our case study. Despite all these extensions, we are still able to preserve the polynomial-time complexity of our tool. Our prototype implementation and case study demonstrate that the runtime of P-Rex is not only of theoretical and asymptotic interest but also relevant in practice.

3 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 be a nonempty set of MPLS labels that appear (possibly arbitrarily nested) in headers of packets of an MPLS network. We define the set Op of allowed MPLS operations on a packet header by $Op = \{swap(\ell) \mid \ell \in L\} \cup \{push(\ell) \mid \ell \in L\} \cup \{pop\}$.

Definition 1 (MPLS Network)**.** An MPLS network is a tuple $N = (V, I, L, E, \tau)$ 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 $in = in'$,
- $L = M \uplus M^{\perp} \uplus L^{IP}$ is the set of the label stack symbols where M is the MPLS label set, $M^{\perp}$ is the set of MPLS labels with the bottom of the stack bit set to true and L^{IP} is a set of labels for IP routing information, and

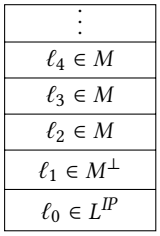

Figure 1: A valid label-stack header

- $\tau : I \times L \to (2^{I \times Op^*})^*$ is the global routing table. For every incoming interface and a top-most label, it returns a sequence (representing priorities in case of link failures) of traffic engineering groups that contain pairs: an outgoing interface and a sequence of MPLS operations to be performed on the packet header. It holds that for any input interface, the corresponding output interface must belong to the same router and hence the global routing table can be represented as a collection of local routing tables $\tau_v : I_v \times L \to (2^{I_v \times Op^*})^*$ for each router $v \in V$.

We fix a set F where $F \subseteq E$ of failed links between interfaces. An interface $in \in I$ is *active* if there is an interface $in' \in I$ such that both $(in, in') \in E \setminus F$ and $(in', in) \in E \setminus F$. In other words, we assume that if a link from (in, in') fails then also the link from (in', in) is down.

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. Additionally, MPLS labels contain a stacking bit such that the first MPLS label just above the IP-label has this bit set to true and all other MPLS labels stacked above have this bit set to false. The structure of a valid label-stack header is illustrated in Figure 1. This is formalized in the following definition.

Definition 2 (Valid Header)**.** For a given network $N = (V, I, L, E, \tau)$ we define the set of valid headers $H \subseteq L^*$ by $H = L^{IP} \cup \{\alpha \ell_1 \ell_0 \mid \alpha \in M^*,\ \ell_1 \in M^{\perp},\ \ell_0 \in L^{IP}\}$.

The MPLS operations manipulate the label-stack header by switching out the top-most label (letf-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* $\mathcal{H} : H \times Op^* \hookrightarrow H$ is defined by (where $ops, ops' \in Op^*$, $\ell \in L$, $h \in H$ and ϵ is the empty sequence of operations):

$$\mathcal{H}(\ell h, ops) =$$

$$\begin{cases} \ell h & \text{if } ops = \epsilon \text{ and } \ell \in L, \\ \mathcal{H}(\ell' h, ops') & \text{if } ops = swap(\ell') \circ ops' \text{ and } \ell' h \in H, \\ \mathcal{H}(\ell' \ell h, ops') & \text{if } ops = push(\ell') \circ ops' \text{ and } \ell' \ell h \in H, \\ \mathcal{H}(h, ops') & \text{if } ops = pop \circ ops' \text{ and } \ell \in M \cup M^{\perp}, \\ undefined & \text{otherwise.} \end{cases}$$

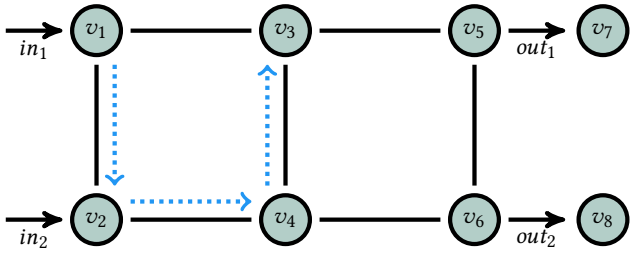

Figure 2: A network example with a failover tunnel

We observe that for any $h \in H$ and any $ops \in Op^*$ we always have $\mathcal{H}(h, ops) \in H$ (provided that $\mathcal{H}(h, ops)$ is defined). In order words the header rewrite function preserves the valid structure of the label-stack symbols, otherwise it is undefined. As an example let $M = \{10, 20, 30\}$, $M^\perp = \{10^\perp, 20^\perp, 30^\perp\}$ and $L^{IP} = \{ip0, ip1\}$. Then $\mathcal{H}(20 \circ 10^\perp \circ ip1,\ pop \circ swap(20^\perp) \circ push(30) \circ push(10)) = 10 \circ 30 \circ 20^\perp \circ ip1$ whereas $\mathcal{H}(20 \circ 10^\perp \circ ip1,\ pop \circ swap(30) \circ push(10)$ is undefined as the expected outcome $10 \circ 30 \circ ip1 \notin H$ is not a valid header (a label with a bottom of the stack symbol was swapped with a label that does not have this bit set).

3.1 Network Example

In Figure 2, we provide an example of a simple network with eight routers $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$ and depicted links: the interfaces take the form $v^{v'}$, denoting an interface of router v that is connected by a link $(v', v) \in E$, with the interface v'^{v} of the router v' and where the labels $L = M \uplus M^\perp \uplus L^{IP}$ consist of

- $M^\perp = \{10, 11, 20, 21, 30, 31, 32, 40, 41\}$,
- $M = \{101, 102, 201, 202, 211, 212, 221, 222\}$, and
- $L^{IP} = \{ip_{out_1}, ip_{out_2}\}$.

The routing table τ for our example network is given in Table 2 and there are no rules for the routers v_7 and v_8, 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 interface then one such rule will be (nondeterministically) applied. If all output interfaces 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 to the network is given by means of network traces.

Router	In I_v	Label	Priority	Out I_v	Operation
v_1	$v_1^{in_1}$	ip_{out_1}	1	$v_1^{v_3}$	push(10)
	$v_1^{in_1}$	ip_{out_2}	1	$v_1^{v_3}$	push(20)
	$v_1^{in_1}$	ip_{out_1}	2	$v_1^{v_2}$	push(10) ∘ push(101)
	$v_1^{in_1}$	ip_{out_2}	2	$v_1^{v_2}$	push(20) ∘ push(201)
v_2	$v_2^{in_2}$	ip_{out_1}	1	$v_2^{v_4}$	push(30)
	$v_2^{in_2}$	ip_{out_2}	1	$v_2^{v_4}$	push(40)
	$v_2^{v_1}$	101	1	$v_2^{v_4}$	swap(102)
	$v_2^{v_1}$	201	1	$v_2^{v_4}$	swap(202)
v_3	$v_3^{v_1}$	10	1	$v_3^{v_5}$	swap(11)
	$v_3^{v_4}$	10	1	$v_3^{v_5}$	swap(11)
	$v_3^{v_1}$	20	1	$v_3^{v_4}$	swap(21)
	$v_3^{v_4}$	31	1	$v_3^{v_5}$	swap(32)
	$v_3^{v_4}$	211	1	$v_3^{v_5}$	swap(212)
	$v_3^{v_4}$	221	1	$v_3^{v_5}$	swap(222)
v_4	$v_4^{v_3}$	21	1	$v_4^{v_6}$	swap(22)
	$v_4^{v_2}$	30	1	$v_4^{v_3}$	swap(31)
	$v_4^{v_2}$	40	1	$v_4^{v_6}$	swap(41)
	$v_4^{v_2}$	102	1	$v_4^{v_3}$	pop
	$v_4^{v_2}$	202	1	$v_4^{v_3}$	pop
	$v_4^{v_3}$	21	2	$v_4^{v_3}$	swap(22) ∘ push(211)
	$v_4^{v_2}$	40	2	$v_4^{v_3}$	swap(41) ∘ push(221)
v_5	$v_5^{v_3}$	11	1	$v_5^{out_1}$	pop
	$v_5^{v_3}$	31	1	$v_5^{out_1}$	pop
	$v_5^{v_3}$	222	1	$v_5^{v_6}$	pop
	$v_5^{v_3}$	212	1	$v_5^{v_6}$	pop
v_6	$v_6^{v_4}$	22	1	$v_6^{out_2}$	pop
	$v_6^{v_4}$	41	1	$v_6^{out_2}$	pop

Table 2: Routing table for the network from Figure 2

3.2 Network Traces

Let us fix a network $N = (V, I, L, E, \tau)$ 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 input interfaces (that uniquely identify the routers that received the packet) together with the corresponding label-stack header the packet arrived with at each router.

Before we give the formal definition of a trace, we shall fix some notation. Let $\tau(in, \ell) = O_0 O_1 \dots O_n$ where each $O \in \{O_0, \dots, O_n\}$ is a traffic engineering group $O = \{(in_1, ops_1), (in_2, ops_2), \dots, (in_m, ops_m)\}$ consisting of output interfaces and sequences of MPLS operations such that the group O_0 has a higher priority than O_1, and O_1 has a higher priority than O_2 and so on. By $I(O) = \{in_1, in_2, \dots, in_m\}$ we denote the set of all interfaces that appear in the traffic engineering group O and we call such a group *active* if there is at least one i, $1 \le i \le m$, such that the interface in_i is active (i.e. there is $in' \in I$ such that $(in_i, in') \in E \setminus F$).

Finally, we define a function that returns the set of all active rules in the sequence of a traffic engineering groups. At the same time we change the outgoing interfaces with the incoming ones in the next hop, as follows: $\mathcal{A}(O_0 O_1 \dots O_n) = \{(in', ops) \mid (in, ops) \in O_j$ such that in is an active interface and $(in, in') \in E\}$ where j is the lowest index such that O_j is an active traffic engineering group. If no such j exists then $\mathcal{A}(O_0 O_1 \dots O_n) = \emptyset$.

Definition 4 (Network Trace)**.** A *trace* in a network $N = (V, I, L, E, \tau)$ with the set $F \subseteq E$ of failed links is any finite sequence $(in_1, h_1), (in_2, h_2), \dots, (in_n, h_n)$ of interface-header pairs from $I \times H$

where for all i, $1 \leq i < n$, we have $h_{i+1} = \mathcal{H}(h_i, ops)$ for some $(in_{i+1}, ops) \in \mathcal{A}(\tau(in_i, head(h_i))$ where $head(h_i)$ is the top-most label of h_i.

The network routing table from Table 2 encodes four label switched paths from the interfaces in_1 and in_2 to either out_1 and out_2, depending on the destination IP address. An example of a trace without any failed links ($F = \emptyset$) follows.

$$(v_1^{in_1}, ip_{out_1}), (v_3^{v_1}, 10 \circ ip_{out_1}), (v_5^{v_3}, 11 \circ ip_{out_1}), (v_7^{v_5}, ip_{out_1})$$

In our example network there are two protected links: (v_1, v_3) and (v_4, v_6). To protect these, for each label switching path going through these links we need 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_1 and v_3 fails, i.e. $F = \{(v_1, v_3)\}$, then we get the following trace instead

$$\begin{gathered}(v_1^{in_1}, ip_{out_1}),\\ (v_2^{v_1}, 101 \circ 10 \circ ip_{out_1}),\\ (v_4^{v_2}, 102 \circ 10 \circ ip_{out_1}),\\ (v_3^{v_4}, 10 \circ ip_{out_1}),\\ (v_5^{v_3}, 11 \circ ip_{out_1}),\\ (v_7^{v_5}, ip_{out_1})\end{gathered}$$

so that the failed link is tunneled through the routers v_2 and v_4 after which the original label switching path is restored.

3.3 A Query Language for MPLS Networks

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

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

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 routers V, describing the (potentially infinite) set of allowed routing traces 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 is given by the abstract syntax

$$a ::= s \mid . \mid [\hat{}s_1, \ldots, s_n] \mid a_1 + a_2 \mid a_1 a_2 \mid a^*$$

where

s	is a symbol from Σ,
$.$	is a wildcard for any symbol from Σ,
$[\hat{}s_1, ..., s_n]$	stands for any symbol $s \in \Sigma \setminus \{s_1, \ldots, s_n\}$,
$a_1 + a_2$	is the choice between a_1 and a_2,
$a_1 a_2$	is the concatenation of a_1 and a_2, and
a^*	is the concatenation of 0 or more occurrences of a.

The set of all regular expressions over Σ is denoted by $Reg(\Sigma)$ and we assume a standard definition of the language $Lang(a) \subseteq \Sigma^*$ that is described by a regular expression a.

We now provide a formal definition of a network query.

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

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

Definition 7. A trace $(in_1, h_1), (in_2, h_2), \ldots, (in_n, h_n)$ in a network $N = (V, I, L, E, \tau)$ with the set F of failed links *satisfies* a query $< a > \quad b \quad < c > \quad k$ if and only if $|F| \leq k$, $h_1 \in Lang(a)$, $h_n \in Lang(c)$ and $v_1 v_2 \ldots v_n \in Lang(b)$ such that $in_i \in I_{v_i}$ for all i, $1 \leq i \leq n$.

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

Problem 1 (Query Satisfiability Problem)**.** Given a network and a query $q = < a > \quad b \quad < c > \quad k$ 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

$$< ip_{out_1} > \quad v_1 \, (.)^* \, v_7 \quad < ip_{out_1} > \quad 0$$

is satisfied due to the existence of a trace that starts with the initial header ip_{out_1} in the router v_1 and reaches in a number of hops the router v_7 with the header ip_{out_1}. The trace is possible without any failed links by visiting the routers v_1, v_3, v_5 and v_7 as demonstrated in Section 3.2. On the other hand the query

$$< ip_{out_1} > \quad v_1 \, (.)^* \, v_4 \, (.)^* \, v_7 \quad < ip_{out_1} > \quad 0$$

is not satisfied, as without any failed links the traffic from the interface $v_1^{in_1}$ with the header ip_{out_1} is never routed though the router v_4, however, the same query which allows one link failure is satisfied as shown by the second trace in Section 3.2. Another query

$$< ip_{out_1} + ip_{out_2} > \quad v_1 \, [\hat{}v_3]^* \, v_7 \quad < (.)^* > \quad 2$$

asks whether, under the assumption that at most two links failed, a packet with the header label ip_{out_1} or ip_{out_2} can, from the router v_1, reach (with arbitrary header) the router v_7 while avoiding the router v_3. This query is not satisfied in our example network.

4 FROM NETWORKS TO AUTOMATA

We shall now explain how to reduce the query satisfability 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.

4.1 Preliminaries

A *nondeterministic finite automaton* (NFA) is a 5-tuple $N = (S, \Sigma, \delta, s_0, s_f)$ 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 $C(N)$ be the set of all such configurations. We define the *transition relation* (using infix notation) $\to_\delta \subseteq C(N) \times C(N)$ 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 N if $(s_0, w) \to_\delta^* (s_f, \epsilon)$. We denote the set of all accepted strings by $Lang(N)$.

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 $\to_\lambda^*$.

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

Theorem 1 ([7, 11])**.** 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. The question whether $(q_0, h_0) \to_\lambda^* (q, h)$ is decidable in polynomial time.

4.2 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 2. [25] Given a regular expression $a \in Reg(\Sigma)$ we can construct in linear time an equivalent NFA $N = (S, \Sigma, \delta, s_0, s_f)$ such that $Lang(N) = Lang(a)$.

Let $w = w_0 w_1 \dots w_n$ be a string. The reverse of w is defined as $w^R = w_n w_{n-1} \dots w_0$. 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 3. [6] Given an NFA $N = (S, \Sigma, \delta, s_0, s_f)$, we can in linear time construct an NFA N^R recognizing the reverse language $Lang^R(N)$.

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 $N = (Q, \Sigma, \delta, q_0, q_f)$, we define the *destructing* PDA $P_d = (Q, \Gamma, \lambda, q_0, q_f)$ such that $\Gamma = \Sigma \cup \{\bot\}$ where $\bot$ is the symbol for the bottom of stack and the transition function λ includes $(q', \epsilon) \in \lambda(q, \ell)$ for every $q, q' \in Q$ and $\ell \in \Sigma$ such that $q' \in \delta(q, \ell)$, and $(q', \ell) \in \lambda(q, \ell)$ for every $q, q' \in Q$ and $\ell \in \Gamma$, such that $q' \in \delta(q, \epsilon)$. It is easy to observe that the destructing pushdown has the following property.

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

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. However, as the top of the stack is never tested but we still need an extra rule for every possible top of the stack, this would unnecessarily create a large number of rules (and our experiments show a large performance penalty). So we instead suggest an alternative construction that considerably reduces the number of needed pushdown rules. The intuition is to push a new symbol (we use $*$ in our case) on the top of the stack instead and in each step perform the swap operation for the actual symbol together with pushing again the symbol $*$ on top of the stack.

Let N be an NFA and let $N^R = (S, \Sigma, \delta^R, s_0, s_f)$ be an NFA that recognizes (by Theorem 3) the reverse language of N such that $Lang(N^R) = Lang^R(N)$. The *constructing* PDA $P_c = (Q, \Gamma, \lambda, q_0, q_f)$ is defined by $Q = S \cup \{q_0, q_f\}$, where q_0 and q_f are unique start and end states such that $q_0, q_f \notin S$, $\Gamma = \Sigma \cup \{\bot, *\}$ where $\bot$ and $*$ are fresh stack symbols such that $\bot, * \notin \Sigma$, and the transition function λ consists of the rules: $(s_0, *\bot) \in \lambda(q_0, \bot)$, $(q_f, \epsilon) \in \lambda(s_f, *)$, $(q', *\ell) \in \lambda(q, *)$ for every $q, q' \in Q$ and $\ell \in \Sigma$ such that $q' \in \delta^R(q, \ell)$, and $(q', *) \in \lambda(q, *)$ for every $q, q' \in Q$ such that $q' \in \delta^R(q, \epsilon)$.

Now we can formulate the expected theorem.

Theorem 5. Given an NFA $N = (S, \Sigma, \delta, s_0, s_f)$, the constructed PDA $P_c = (Q, \Gamma, \lambda, q_0, q_f)$ has a computation $(q_0, \bot) \to_\lambda^* (q_f, h\bot)$ if and only if $h \in Lang(N)$.

Finally, we describe a construction that allows us to restrict the possible executions of a pushdown automaton only to those where the sequence of visited control states belongs to a given regular language (represented by an NFA). The idea is to synchronize the execution of the pushdown and the NFA via a synchronized product of the two automata.

Definition 8 (Synchronized Pushdown)**.** Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and an NFA $N = (S, \Sigma, \delta, s_0, s_f)$ where $\Sigma = Q$, we construct the PDA $P' = (Q', \Gamma', \lambda', q_0', q_f')$ where

- $Q' = (Q \times S) \cup (\{start\} \times S)$ such that $start \notin Q$,
- $\Gamma' = \Gamma$,
- $q_0' = (start, s_0)$ is the initial state,
- $q_f' = (q_f, s_f)$ is the final state, and
- $\lambda' : Q' \times \Gamma' \to 2^{Q' \times \Gamma'^*}$ is the transition function defined by the following three rules.

For every $(q, s), (q, s') \in Q'$ and $\ell \in \Gamma'$

a) P' contains the rule

$$((q, s'), \ell) \in \lambda'((q, s), \ell)$$

whenever $s' \in \delta(s, \epsilon)$,

b) P' contains the rule

$$((q', s'), \alpha) \in \lambda'((q, s), \ell)$$

whenever $(q', \alpha) \in \lambda(q, \ell)$ and $s' \in \delta(s, q')$, and

c) P' contains the rule

$$((q_0, s'), \ell) \in \lambda'((start, s), \ell)$$

for every $s, s' \in S$ such that $s' \in \delta(s, q_0)$.

Rule a) allows us to perform ϵ-steps in the NFA without affecting the PDA configuration. Rule b) encodes that if the PDA has a transition that changes the state from q to q' and the NFA in the state s can read the symbol q' and reach the state s', then both automata can perform this move in a lockstep. Finally, Rule c) allows us to start the computation by reading the initial state q_0 and removing the *start* state. The following theorem formalizes the idea of the construction.

Theorem 6. There is a computation in the pushdown $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_{n-1}, h_{n-1}) \to_\lambda (q_f, h_n)$ where $q_0 q_1 \dots q_{n-1} q_f \in Lang(N)$ if and only if $((start, s_0), h_0) \to^*_{\lambda'} ((q_f, s_f), h_n)$.

4.3 Removal of Redundant PDA Rules

A typical network can contain a large number of routing table entries with many different labels. For example, the network used in our case study produces 935,045 rules in our MPLS network model. Once the corresponding pushdown automaton is synchronized with even the simplest NFA representing a basic reachability query $\langle\rangle$ $(.)^*$ $\langle\rangle$, we end up with an automaton with 86,256,450 transitions and a file size of 4.4 GB. However, most of these transitions will be redundant (they cannot be applied to any reachable pushdown configuration). In what follows, we develop an efficient technique that will allows us to over-approximate the set of possible symbols on top of the stack in a given control state and remove a significant portion of redundant pushdown transition. This reduces the number of pushdown transitions in our example to 17,847,465 and the resulting pushdown can be stored in a file of size 875 MB.

We first compute the function `find_tops(`P, ℓ_0`)` presented in Algorithm 1 and over-approximate the top of the stack symbols in all reachable configurations of the pushdown P starting in the initial configuration (q_0, ℓ_0). The algorithm returns a function T 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 $(q_0, \ell_0) \to^*_\lambda (q, \ell w)$ then $\ell \in T[q]$.

We can now use the computed function T that approximates the possible symbols on the top of the stack to prune the rules of a given pushdown automaton as follows. Let $P = (Q, \Gamma, \lambda, q_0, q_f)$ be a PDA, ℓ_0 be a stack symbol and T the function returned by the call `find_tops(`P, ℓ_0`)`. We construct a PDA $P' = (Q, \Gamma, \lambda', q_0, q_f)$ such that $\lambda'(q, \ell) = \{(q', \alpha) \in \lambda(q, \ell) \mid \ell \in T[q]\}$. Due to Lemma 1 we can now conclude with our pruning theorem.

```
1  Function find_tops(P, ℓ0)
       Input  : A PDA P = (Q, Γ, λ, q0, qf) and ℓ0 ∈ Γ.
       Result : A function T : Q → 2^Γ.
2      Rules ← {(q, ℓ, q', α) | (q', α) ∈ λ(q, ℓ)};
3      for q ∈ Q do T0[q] ← ∅ ;
4      T0[q0] ← {ℓ0};  n ← 0;
5      repeat
6          n ← n + 1;  Tn ← Tn−1;
7          for (q, ℓ, q', α) ∈ Rules do
8              if ℓ ∈ Tn[q] then
9                  if |α| ≥ 1 then
10                     Tn[q'] ← Tn[q'] ∪ {head(α)};
11                 else
12                     Tn[q'] ← Tn[q'] ∪ Γ;
13                 end
14             end
15         end
16     until Tn = Tn−1;
17     return Tn;
18 end
```

Algorithm 1: Approximation of top of the stack symbols

Theorem 7. There is a computation $(q_0, \ell_0) \to_\lambda (q_1, w_1) \to_\lambda \cdots \to_\lambda (q_n, w_n)$ in P iff there is a computation $(q_0, \ell_0) \to_{\lambda'} (q_1, w_1) \to_{\lambda'} \cdots \to_{\lambda'} (q_n, w_n)$ in P'.

4.4 Encoding MPLS Reachability into 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 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.

4.4.1 Over-approximation. Assume an MPLS network $N = (V, I, L, E, \tau)$ with maximum k link failures. By $\overline{Ops}$ we denote the set of all MPLS operation sequences and all suffixes of such sequences that appear in the routing table τ. We recall that the routing function τ maps an interface and a label to a sequence of traffic engineering groups $\tau(in, \ell) = O_0 O_1 \dots O_n$ where $O_c = \{(in_c^1, ops_c^1), (in_c^2, ops_c^2), \dots, (in_c^{l_c}, ops_c^{l_c})\}$ for all $0 \le c \le n$. We define a k-failure aware routing function by $\tau^k(in, \ell) = \bigcup_{j=0}^{i} O_j$ where i the smallest index such that the cardinality of the set $\{in_1^1, in_1^2, \dots, in_1^{l_1}, in_2^1, in_2^2, \dots, in_2^{l_2}, \dots, in_i^1, in_i^2, \dots, in_i^{l_i}\}$ is larger than k.

For the network $N = (V, I, L, E, \tau)$ we define an over-approximating pushdown automaton $P(N) = (Q, \Gamma, \lambda, q_0, q_f)$ where $Q = \{(in, ops) \mid in \in I, ops \in \overline{Ops}\} \cup \{q_0, q_f\}$, $\Gamma = L$, and λ is defined as follows:

a) $P(N)$ contains the rule

$$((in', ops), \ell) \in \lambda((in, \epsilon), \ell)$$

for every $\ell \in \Gamma$ and every $(out, ops) \in \tau^k(in, \ell)$ such that $(out, in') \in E$.

b) $P(N)$ contains the rule

$$((in, ops'), \ell') \in \lambda((in, ops), \ell)$$

if $ops = swap(\ell') \circ ops'$ where $\ell, \ell' \in M$ or $\ell, \ell' \in M^\perp$ or $\ell, \ell' \in L^{IP}$.

c) $P(N)$ contains the rule

$$((in, ops'), \ell'\ell) \in \lambda((in, ops), \ell)$$

if $ops = push(\ell') \circ ops'$ where $\ell \in M \cup M^\perp$ and $\ell' \in M$, or $\ell \in L^{IP}$ and $\ell' \in M^\perp$.

d) $P(N)$ contains the rule

$$((in, ops'), \epsilon) \in \lambda((in, ops), \ell)$$

if $pop \circ ops'$ and $\ell \in M \cup M^\perp$.

e) $P(N)$ contains the rule

$$((in, \epsilon), \ell) \in \lambda(q_0, \ell)$$

for every $in \in I$ and every $\ell \in L$.

f) $P(N)$ contains the rule

$$(q_f, \ell) \in \lambda((in, \epsilon), \ell)$$

for every $in \in I$ and every $\ell \in L$.

We can now state a key property of our encoding.

Theorem 8. Let $N = (V, I, L, E, \tau)$ be an MPLS network model and let k be the maximum number of link failures. Any trace $(in_0, h_0), (in_1, h_1), \ldots (in_n, h_n)$ in the network where $|F| \leq k$ implies that $(q_0, h_0) \rightarrow_\lambda^* (q_f, h_n)$ in the constructed over-approximating PDA $P(N) = (Q, \Gamma, \lambda, q_0, q_f)$.

Proof. The proof is by noticing that in each control state of the form (in_i, ϵ) we can perform by rule a) the hop (according to the network trace) to the next incoming interface represented by the control state (in_{i+1}, ops) where ops is the chain of MPLS operations to be performed (this directly follows the definition of a next hop in Definition 4). Notice also that the k-failure aware routing function $\tau^k(in, \ell)$ over-approximates the set of active interfaces available for the next hop as defined by the function $\mathcal{A}(O_0 O_1 \ldots O_n)$ for a given set of failed links F where $|F| \leq k$. The application of the header-rewrite function $\mathcal{H}$ defined in Definition 3 is then naturally implemented, step by step, by the execution of the rules b), c) and d) until we reach the control state (in_{i+1}, ϵ) and we are ready to execute the next hop. The rules e) and f) simply allow us to initialize resp. accept the pushdown execution for any available interface. □

4.4.2 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 could 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 can execute a subset of network traces. The intuition is to reuse the over-approximating pushdown construction where we add a third component to the control state (representing a counter of encountered failed links during the routing), so that the control states have the form (in, ops, i) where $0 \leq i \leq k$ such that i is the number of failed links so far. All rules b) to d) simply copy this number i without any change, rule e) sets the counter $((in, \epsilon, 0), \ell) \in \lambda(q_0, \ell)$ so that i is initialized to 0 and rule f) is applicable for any control state where $i \leq k$. The only rule that modifies the value i is the updated rule a) that in the under-approximating PDA looks as follows:

$$((in', ops, i + j), \ell) \in \lambda((in, \epsilon, i), \ell)$$

for every $\ell \in \Gamma$ and every j and $(out, ops) \in \tau^j(in, \ell)$ where $i + j \leq k$ such that $(out, in') \in E$. This rule simply adds 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 $(q_0, h_1) \rightarrow_\lambda ((in_1, ops_1, i_1), h_1) \rightarrow_\lambda ((in_2, ops_2, i_2), h_2) \rightarrow_\lambda \ldots ((in_n, ops_n, i_n), h_n) \rightarrow_\lambda (q_f, h_n)$ in the under-approximating pushdown has a *loop* if there are two indices j_1 and j_2 such that $1 \leq j_1 \neq j_2 < n$ and $in_{j_1}, in_{j_2} \in I_v$ for some $v \in V$, in other words the interfaces in_{j_1} and in_{j_2} belong to the same router v.

Theorem 9. Let $N = (V, I, L, E, \tau)$ be an MPLS network and let k be the maximum number of link failures. If $(q_0, h_0) \rightarrow_\lambda^* (q_f, h_n)$ is a computation without a loop in the under-approximating pushdown $P(N) = (Q, \Gamma, \lambda, q_0, q_f)$ then there is a trace $(in_0, h_0), (in_1, h_1), \ldots, (in_n, h_n)$ in the network for some F such that $|F| \leq k$.

4.5 Solving Query Satisfability Problem

We have now described all the necessary automata constructions and are ready to provide a solution to the problem of query satisfability in a given MPLS network model (that is obtained automatically from the network topology and routing tables). 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 satisfability problem. First, we construct (either using over- or under-approximation) the pushdown $P(N)$ with the property stated in Theorem 8 resp. Theorem 9. This pushdown $P(N)$ is then using the synchronized product construction in Definition 8, and paired with the NFA that describes the allowed router sequences given by the regular expression b in our query. Let us call the resulting pushdown as P^{route} and recall that it satisfies Theorem 6. For the regular expression a we create a constructing PDA P_c that satisfies the property in Theorem 5 and for the expression c we construct

the destructing PDA P_d with the property in Theorem 4. Finally, we combine P^{route}, P_c and P_d into a single pushdown by running first P_c and once it enters its accepting state, we continue with the execution of P^{route} 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 conclude with the main result of our paper.

Theorem 10. Let $N = (V, I, L, E, \tau)$ be an MPLS network and let $q = \langle a \rangle\ b\ \langle c \rangle\ k$ be a query on N. Let $P_{final} = (Q, \Gamma, \lambda, q_0, q_f)$ be the pushdown constructed from N and q.

- Let P_{final} be a pushdown 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 automaton P_{final}.
- If $(q_0, \bot) \rightarrow^*_\lambda (q^f, \bot)$ is reachable by a loop-free path in the automaton P_{final} that is constructed using the under-approximation then there is a trace in the network N that satisfies the query q for some set of failed links F such that $|F| \leq k$.

We can see that the problem of query satisfability in an MPLS network is reduced to a reachability problem in the automaton P_{final}, and thanks to Theorem 1, this problem can be solved in polynomial time. A negative answer to the reachability problem in the over-approximating pushdown 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 implies that the given query is satisfied in the network. Otherwise the answer to the query satisfability problem is inconclusive.

5 IMPLEMENTATION AND EVALUATION

We have implemented a prototype of P-Rex in Python 3.6. In the following, we will report on our experiences and experiments with our prototype, both in synthetic scenarios (in order to be able to compare P-Rex to existing tools) as well as in an industrial case study in collaboration with a network operator NORDUnet. The source code of our tool is available at www.github.com/p-rexmpls and released under an open-source non-commercial license, including the data (in anonymized form) collected during our case study. All our experiments were executed on a 4 CPU NUMA architecture (AMD Opteron Processor 6376), with each CPU having 16 physical cores running at 2.3 Ghz, and a total of 1 TB of memory. We note that our implementation is single threaded but we used the 64 available cores to run several computations concurrently.

5.1 Comparing HSA and P-Rex

We compare the performance of P-Rex to HSA [17], in a series of scalable instances by considering the simple network from Figure 3 on the left and repeatedly replacing the link between v_1 and v_3 by the same subnetwork. 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. The HSA tool simply enumerates all possible sets of failed links and performs a reachability analysis for each such configuration,

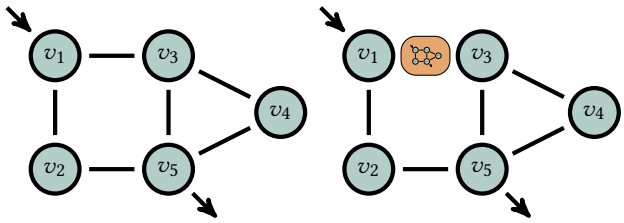

Figure 3: Scaling of our synthetic network

P-Rex HSA	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Nesting: 0	0.6	0.6	0.6	0.6
Routers: 5	0.2	0.1	0.1	0.2
Nesting: 1	0.6	0.6	0.6	0.6
Routers: 10	0.1	0.1	0.4	3.7
Nesting: 2	0.6	0.6	0.6	0.6
Routers: 15	0.1	0.3	1.9	55.9
Nesting: 3	0.6	0.6	0.6	0.6
Routers: 20	0.1	0.3	6.8	335.6
Nesting: 4	0.6	0.6	0.6	0.6
Routers: 25	0.1	0.6	16.4	567.2
Nesting: 5	0.6	0.6	0.6	0.6
Routers: 30	0.1	1.0	34.6	1901.1
Nesting: 6	0.6	0.6	0.6	0.7
Routers: 35	N/A	N/A	N/A	N/A

Table 3: P-Rex vs HSA runtime (in seconds)

whereas P-Rex uses the over-approximation approach: the concrete query is $\langle . \rangle\ v_1(.)^* v_5\ \langle . . \rangle$ which is not satisfied and hence our answer is conclusive. In Table 3 we can see the runtime in seconds for P-Rex (top part of each entry) vs. HSA (bottom part of each entry). A gray box shows the situation where P-Rex is not faster than HSA and white boxes are the instances where we perform better than HSA. We can see that once we scale the number of failed links or the size of the network (and in particular the nesting depth of MPLS labels), the performance of HSA deteriorates significantly. We also note that HSA cannot handle the network with 35 routers due to its internal limitations.

In conclusion, this experiment demonstrates the our tool scales significantly better both in the nesting depth of the MPLS labels as well as the number of considered failed links.

5.2 Industrial Case Study

We next report on a case study we performed on a real-world MPLS network operated by NORDUnet. NORDUnet is a regional service provider and its network consists of 24 MPLS routers, geographically distributed across several countries. The routers are primarily Juniper, running JunOS and their MPLS network uses more than 30,000 labels. In total, the number of forwarding rules collected from this network amounts to almost one million in our model. The forwarding tables format is documented in depth at [15].

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 route forwarding-table family mpls extensive | display .ml`
- `show isis adjacency detail | display .ml`

As JunOS only matches on the incoming top label whereas in our network model we match on the incoming interface as well as the incoming top label, P-Rex adds the forwarding entries for all interfaces. After retrieving this information, an automatic script is run to generate the XML code which is the input to the next stage of constructing the MPLS model. The resulting pushdown automaton before the synchronization contains 1.8 million transitions. After the lockstep construction, the number of transitions can grow to 20 million and this input is forwarded to Moped tool.

5.2.1 Reachability Matrix. In order to check the feasibility of our approach on the network provided by NORDUnet, we use our cluster to compute the whole reachability matrix between any pair of routers (router names are anonymized by using letters A to X) in the network as showed in Table 4. The verified query $<.>\ Y\ (.)^*\ Z\ <.>\ 2$ asks whether a packet with some IP-label only on the top of the stack can be forwarded from one router to another, assuming at most 2 link failures. It took the average time of 1 hour and 1 minute to answer such a query for a given pair of routers and the table documents that in almost all cases we are able to provide a conclusive answer (we use ✓ for the positive answer and . for the negative one). Only in three cases the answer was inconclusive (denoted by the question mark).

The reader may observe that only five routers are connected in the reachability matrix. The reason is that the operator implements IP-based routing only on a few selected ingress routers. Most of the traffic from other networks is forwarded though a number of MPLS service labels that are on top of the IP-address when a packet arrives from a neighbouring network. For this reason, we run another experiment with the query $<..>\ Y\ (.)^*\ Z\ <(.)^*>\ 2$ that allows the input header contain two labels (the service label followed by the IP label). The resulting reachability matrix for this case is shown in Table 5. We can notice a significant increase in the connectivity among the routers and this confirms the information we got from NORDUnet about the heavy use of service labels in their network. We notice a slight increase in the number of inconclusive answers, likely due to the fact that the routers are reachable but exhibit a looping behavior for which our under-approximation does not provide a conclusive answer. Nevertheless, the number of inconclusive answers is below 3% of all asked queries and we evaluate the performance and applicability of our tool to this case as convincing.

5.2.2 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 was interested in the question, whether a packet arriving with the service label 234 on top of some IP-label at router A, can be routed to router B while visiting router F such that the service label is popped. The

Table 4: Reachability matrix for $<.>\ Y\ (.)^*\ Z\ <.>\ 2$

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

Table 5: Reachability matrix for $<..>\ Y\ (.)^*\ Z\ <(.)^*>\ 2$

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

P-Rex query is formulated as

$$< 234\ .\ >\ A\ (.)^*\ F\ (.)^*\ B\ <>\ 0$$

and in case of no failed links, out tool was able to conclude in 38m 26s while using 7.00 GB of memory that this is not the case. However, the same query with $k = 1$ holds as a packet from A to B can indeed be routed through the router F, in case of one link failure, and our tool returns in 91m 14s a trace demonstrating such a routing. Another interesting question is whether it is possible that a packet arriving to some router with the service label 234 will do 3 or more hops in the network. The corresponding query

$$< 234\ .\ >\ .\ .\ .\ (.)^*\ <>\ 0$$

does not hold (the answer was computed in 48m 44s using 6.01 GB of memory) as there are at most two hops for every such packet. However, in case of one link failure, it is possible that the packet makes at least 5 hops as showed by the query

$$< 234\ .\ > \ldots\ldots (.)^* <>\ 1$$

that was answered by our tool positively in 109m 32s using 14.18 GB of memory, while at the same time returning a valid network trace. Finally, the network operator claims that the service label 800 is never popped as it is supposed to define a tunnel through the network and this is indeed the case as the corresponding query

$$< 800\ .\ > (.)^* < [\hat{}800]\ .\ >\ 1$$

has a negative answer (computed in 28m 9s using 5.04 GB of memory), even in the case of one failed link. These examples demonstrate a wide applicability of our technique that allows us to answer a variety of complex network questions.

6 CONCLUSION

While there is 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 specific case of MPLS networks, there exist techniques that allow for a fast, polynomial-time analysis, even accounting for a seemingly exponential number of possible failure configurations. In particular, we presented a what-if analysis tool P-Rex that significantly outperforms existing tools, and we reported 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 regards the synthesis [8, 9] 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. We also thank Peter Gjøl Jensen and Mads Boye for their assistance with getting access to AAU model checking cluster, as well as our shepherd Hongqiang Liu.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.* 49, 1 (2014), 113–126.

[2] Alia K Atlas and Alex Zinin. 2008. Basic specification for IP fast-reroute: loop-free alternates. IETF RFC 5286.

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication ((SIGCOMM)'17)*. ACM, 155–168. https://doi.org/10.1145/3098822.3098834

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proc. ACM SIGCOMM*. ACM, 155–168.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proc. ACM SIGCOMM*. ACM, 328–341.

[6] J.A. Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata* 12 (1962), 529–561.

[7] J.R. Büchi. 1964. Regular canonical systems. *Arch. Math. Logik u. Grundlagenforschung* 6 (1964), 91–111.

[8] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis. In *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 261–281.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 579–594.

[10] Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. 2014. IP fast rerouting for multi-link failures. In *Proc. IEEE INFOCOM*. ACM, 2148–2156.

[11] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. 12th International Conference on Computer Aided Verification (CAV) (LNCS)*, Vol. 1855. Springer, 232–247.

[12] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis using a Succinct Control Plane Representation. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 217–232.

[13] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A coalgebraic decision procedure for NetKAT. In *ACM SIGPLAN Notices*, Vol. 50 (1). ACM, 343–355.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference ((SIGCOMM)'16)*. ACM, 300–313. https://doi.org/10.1145/2934872.2934876

[15] Juniper. 2018. Show Route Forwarding-Table. Technical Documentation https://www.juniper.net/documentation/en_US/junos/topics/reference/command-summary/show-route-forwarding-table.html.

[16] David M Kahn. 2017. Undecidable Problems for Probabilistic Network Programming. In *MFCS'17*, Vol. 83. LIPIcs-Leibniz International Proceedings in Informatics, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 1–16.

[17] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proc. 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 113–126.

[18] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 15–27.

[19] Kim G. Larsen, Stefan Schmid, and Bingtian Xue. 2016. 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, 1–18.

[20] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, Vol. 41 (4). ACM, 290–301.

[21] Michael Menth, Michael Duelli, Ruediger Martin, and Jens Milbrandt. 2009. Resilience analysis of packet-witched communication networks. *IEEE/ACM transactions on Networking (ToN)* 17, 6 (2009), 1950–1963.

[22] RFC 8001. 2017. RSVP-TE Extensions for Collecting Shared Risk Link Group (SRLG). https://datatracker.ietf.org/doc/rfc8001/.

[23] Stefan Schmid and Jiří Srba. 2018. Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks. In *IEEE International Conference on Computer Communications (INFOCOM'18)*. IEEE, 1–9.

[24] Stefan Schwoon. 2002. *Model-Checking Pushdown Systems*. Ph.D. Thesis. Technische Universität München. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf

[25] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[26] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2012. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking (ToN)* 20, 6 (2012), 1814–1827.

[27] Dahai Xu, Yizhi Xiong, Chunming Qiao, and Guangzhi Li. 2004. Failure protection in layered networks with shared risk link groups. *IEEE Network* 18, 3 (2004), 36–41.

[28] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. 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, 87–99.