Prex: A tool for Reachability Analysis In MPLS Networks

Jesper Stenbjerg Jensen, Troels Beck Krøgh
Jonas Sand Madsen, and Marc Tom Thorgersen
Department of Computer Science, Aalborg University
{jejens13,tkragh13,jsma13,mthorg13}@student.aau.dk

June 8, 2018

Abstract

We describe Prex, a tool for analyzing MPLS networks leveraging an automata theoretic approach. While we base our approach on previous works, we extend their approaches in a multitude of ways. We introduce a method of constructing queries based on regular expressions which permits the user to write highly complex queries, and implement a method of verifying these queries on the networks. We benchmark our tool and compare it with another recognized tool. We perform a case study on a large scale in production MPLS network, and use our tool verify certain interesting properties in their network.

Summary

We include this summary since it is a formal requirement by Aalborg University. In Section 1 on the following page we introduce the background and related work. First we provide the reader with a thorough amount of background information about Multiprotocol Label Switching (MPLS), and MPLS fast reroute. Then we introduce Segment Routing (SR), which is an emerging technology within networking. SR is interesting to us since it can be deployed on top of an existing MPLS infrastructure with no changes to the data-plane. We present a large amount of related work to show the cutting edge of the current knowledge within this field.

We then define our central working problem, and an overview of how were planning to solve it. We then list our contributions in the paper.

In Section 2 on page 12 we introduce our MPLS network model and related definitions. First we model the MPLS operations, and the MPLS network. We then define the set of valid headers and the header rewrite function. Here we also define our query language and traces through the network, and from these construct the main problem we aim to solve in this paper. To give the reader a further understanding of our network model we give a network example, and relate it to all the other concepts defined in this section.

In Section 3 on page 17 we introduce the reader to the automata theory we use for our approach. This includes both well known results and novel results. First we define the non-deterministic finite automata (NFA) and the pushdown automata (PDA). Here we introduce the novel concept of augmented pushdown construction, which computes an NFA and a PDA in lockstep. We show how we construct the over- and under-approximating PDAs for a given MPLS network.

In Section 4 on page 30 we present two generic optimizations to reduce the size of a PDA such that reachability analysis can be performed in less time and using less memory. The first optimization removes a large amount of transitions from the PDAs, in particular when no-op behavior occurs. The second optimization utilized a swap-push operation to reduce the number of transitions in the PDa when the same behavior is desired for all labels, this is used for optimizing the generation of headers.

In Section 5 on page 35 we present how the final PDA is constructed. That is, the PDA which combines all the theory we have and is finally sent to the PDA reachability analysis tool we are utilizing.

Lastly in Section 6 on page 36 we compare the performance of Prex with a recognized tool from the related work. In addition to this, we perform a case study of a real world network provided by NORDUnet.

1 Introduction

In conventional Internet Protocol (IP) network[1, 2], each router needs to analyze each packets header, and find the longest prefix match for the destination IP in its routing tables, from that it can forward the packet to another router. However, for some network types, avoiding the cost of a longest prefix match, and at the same time gaining more control over the traffic flow in the network, is desired. This is one of the core improvements given by Multiprotocol Label Switching (MPLS)[2].

1.1 Background

1.1.1 Multiprotocol Label Switching (MPLS)

In MPLS networks, the ingress router analyzes the packet, and decides its path through the MPLS network. How this path is chosen, and which factors are considered, are up to the implementation. From here it will choose a path for it to take through the network. In MPLS this is achieved by adding a header to each packet, this header is a stack of "label stack entries" each of which is 32 bits. The MPLS label is the most significant, and takes up 20 bits of the 32 bits.

The label is what each router in the network has to analyze before it can forward the packet, according to its MPLS routing table. Before forwarding the packet, the router will typically replace the label with another one in an operation called: swap. This forms the basis for Label Switched Paths (LSPs), which are tunnels through the MPLS network. The ingress router selects an LSP for the packet by pushing an initial label to the label stack, this initial label will be swapped to different labels by each hop in the LSP, and finally it will be popped by the egress router. This forwarding scheme is simple and fast in practice. For MPLS to function it requires precise configuration, significantly the LSPs needs to be determined, for this task it is common to use a method such as Resource Reservation Protocol - Traffic Engineering (RSVP-TE)[3].

MPLS Fast Reroute During normal operation of a network, it is expected that occasionally parts of said network, such as a link, will fail. Since packet loss and lack of connectivity during such events are undesired MPLS has fast reroute features. The purpose of fast reroute is to route traffic around failed elements, in this paper we will focus on failed links. Adding fast reroute functionality for link-protection to an MPLS network, is done through the use of backup tunnels. A backup tunnel is an LSP which has the same source and destination as the original link, when such a backup tunnel exists one can consider the link protected. When the source to a link detects the failure, it will reroute through the backup tunnel. It does so by pushing a label onto the stack, which corresponds to the LSP of the backup tunnel for the link. This operation can be performed recursively if multiple links fail at the same time. This feature is also sometimes known as MPLS local protection[4], since it does not consider the entire topology when rerouting.

To further understand this, consider the following example as shown in Fig. 1. Here the dotted blue line is the normal LSP from *in* on r_1 to *out* on r_7. The link between r_2 and r_3 is broken, and marked as dashed in the figure. r_2 will

then utilize the MPLS fast reroute feature, to use the backup tunnel protecting the broken link. In this example the tunnel does via r_5 and r_6 to its original destination of r_3, then the backup tunnel is completed and the original LSP is active again.

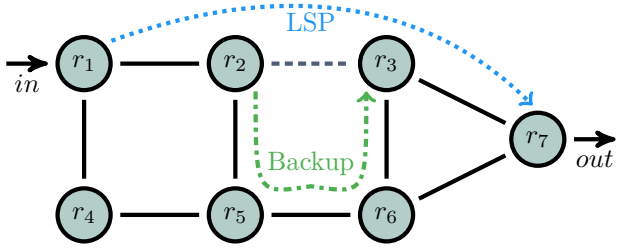


Figure 1: MPLS Fast Reroute example.

1.1.2 Segment Routing

Segment Routing (SR) is part of the source packet routing paradigm of networking[5, 6]. The Source Packet Routing in Networking (SPRING) paradigm gives additional control to network operators compared to traditional shortest-path based methods, wrt. traffic engineering, load-balancing etc.[7]

Segments In SR this control is exposed through a list of instructions for each packet, called segments, hence the name Segment Routing. Only one of the segments is active at any one time, this segment is called the "Active Segment". A segment is an instruction for the router to forward such as: 1. forward the packet along a precomputed (often shortest) path to some destination, 2. forward the packet through a specific interface, and 3. deliver the packet to a specific router or set of routers, such as a firewall.

Segments are often referred to by their Segment Identifier (SID), and the two terms are occasionally used interchangeably even though it may overlook a translation between the two.

Interior Gateway Protocol Segments Each SR-node advertises its prefixes and adjacencies to the Segment Routing Domain, which is the set of nodes participating in the network. This is used by a link-state Interior Gateway Protocol (IGP) protocol to enable expression of any path through the network. There are three proposals for link-state IGP with SR: 1. Intermediate System to Intermediate Systems (IS-IS)[8], 2. Open Shortest Path First (OSPF)[9], and 3. Open Shortest Path First version 3 (OSPFv3)[10].

IGP-Prefix segments identifies a specific path in the SR domain. That is, any packet with its active segment as a IGP-Prefix segment, will follow a path predetermined by an algorithm. In the standard draft[6] two such algorithms are provided: 1. Shortest Path, and 2. Strict Shortest Path. The key difference is that with the shortest path algorithm, local policies may override the predetermined path. An IGP-Node segment simply describes a node, e.g. router, in the network. An IGP-Anycast segment instructs the packet forwarding to the closest node in the anycast set. An anycast set is a

predetermined set of nodes in the network. An IGP-Adjacency segment instructs a node to use a specific link or a set of links, which could be load-balanced by the node. Furthermore a node can advertise a weight for each link in a set of links, with Adjacency segments, this weight can be used for load-balancing through parallel adjacencies.

1.1.3 Operations in SR

To implement these instructions the routers in a SR network have three operations closely resembling the ones found in MPLS.

- PUSH: Insert a segment on top of the segment list as a new active segment,
- NEXT: Remove the active segment (since it has been completed), and
- CONTINUE: Do not change the segment list.

SR is currently under standardization[6] as an RFC under Internet Engineering Task Force (IETF).

The SR standard draft proposes two architectures to implement SR on: 1. MPLS called Segment Routing over MPLS (SR-MPLS), and 2. Internet Protocol version 6 (IPv6) called Segment Routing over IPv6 (SRv6). This paper focuses on MPLS and SR-MPLS.

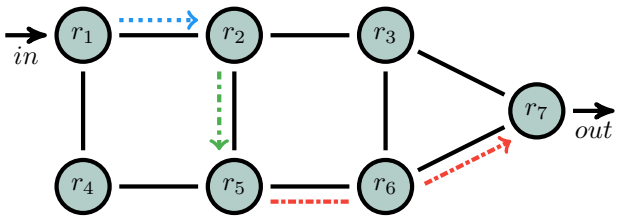


Figure 2: Segment Routing network, with route-trace. The colored lines each represents a segment.

Position	Type	Content	Style on figure
1st	Node	r_2	
2nd	Adjacency	r_2-r_5	
3rd	Node	r_7	

Table 1: Segment List at the start of routing.

R	Active Segment Type	Con.	Op	Comment
r_1	Node	r_2	CONT	Forward to r_2 via link r_1-r_2
r_2	Node	r_2	NEXT	Remove AS, since AS is reached
r_2	Adj	r_2-r_5	NEXT	Remove AS, forward via r_2-r_5
r_5	Node	r_7	CONT	Forward to r_6 via link r_5-r_6
r_6	Node	r_7	CONT	Forward to r_7 via link r_6-r_7
r_7	Node	r_7	NEXT	Remove AS, since AS is reached
r_7	N/A		N/A	None, routing is complete

Table 2: Trace of actions taken during the routing, under normal SR operation.

Example of Segment Routing in action In order to give a more intuitive sense of how SR works, we present the following example. In Fig. 2 we present a SR network of 7 routers, named r_1 through r_7. In the example we have some traffic entering r_1 with the destination of r_7.

With some traffic engineering constraints in mind r_1 constructs a list of segments to lead the traffic to r_7. First it adds a node segment with the SID of r_2, i.e. the first active segment. Then it adds an adjacency segment, which specifies the link from r_2 to r_5. Lastly it adds a node segment for r_7, which is the final destination. The segment list can more easily be visualized as a table:

When the packet is forwarded through the network, it will follow the sequence of events given in Table 2. In the table we have the router which is operating on the packet. Then we have the active segment, which is two parts: firstly which type of segment it is, and secondly what the content of the segment is. Lastly we have what action the router performs, both regarding the segment list and where to send the packet.

1.1.4 Segment Routing Fast Reroute

SR comes with a native Fast Reroute (FRR) technology which relies on Toplogy-Independent Loop-Free-Alternate (TI-LFA) [11]. TI-LFA is similar to its namesake remote Loop-Free-Alternate (rLFA), although where rLFA leverages Label Distribution Protocol (LDP), TI-LFA leverages SR [12]. Leveraging LDP does not grant 100 % topology coverage to rLFA, on the other hand leveraging SR as TI-LFA does grants 100 % topology coverage due to enhanced Traffic Engineering (TE) capabilities through segments. We use the definiton of topology coverage from [13] which means that 100 % topology covers means that reachability is always preserved unless a network bisection occurs, by links failing. Furthermore TI-LFA also guarantees path optimality, something that RSVP-TE does not.

In case of a link failure RSVP-TE FRR would use an established detour LSP set to merge with the original LSP downstream, such as the next-hop router [14]. Such an approach may produce inefficient paths, e.g., by causing congestion due to not considering bandwidth reservations, or by creating a needlessly long path to reach the next-hop router. TI-LFA would create an alternate post-convergence path, using SR segments to reach a node from where the IGP path does not traverse the failed link, from that link the IGP route would be followed, rather than attempt to follow the original path as is the case with RSVP-TE.

Per the SR standard, the introduction of SR to an MPLS network does not cause any change in the MPLS data-plane[15]. Per the TI-LFA standard, TI-LFA prepares data-plane switch-overs activated on failure in similar fashion to RSVP-TE [12]. As such, while the FRR switch-over paths would be different, they would still exist in the same fashion as with RSVP-TE and therefore introduce no real change to the MPLS data-plane. Furthermore introducing TI-LFA is by no means a requirement, the FRR handling can remain the same if so desired, TI-LFA simply leverages SR for benefits, and is recommended by Cisco [11].

It should be noted that TI-LFA only handles a single link failed, and would the require re-convergence before being able to handle another one. However there are attempts to generalize it to multiple link failures, such as [16].

Example of TI-LFA To get a more practical understanding of how TI-LFA works, consider the example from prevously in Fig. 1, where we provided an example of MPLS FRR link-protection. This has the disadvantage of not being a

shortest path, therefore it can cause more congestion than necessary. Under TI-LFA other solutions are possible, significantly it will typically provide a shortest path from the point of failure to the original destination. An example of this is given in Fig. 3.

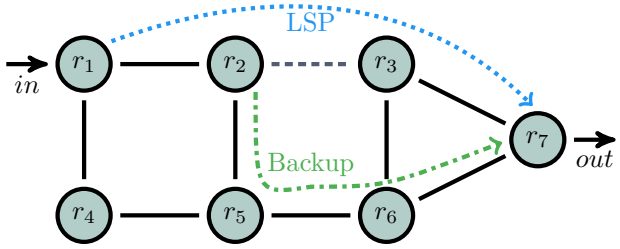


Figure 3: TI-LFA Fast Reroute example.

1.1.5 Segment Routing with MPLS data plane

SR-MPLS can be deployed using existing MPLS hardware and infrastructure. SR is designed in such a way that many of its concepts maps to MPLS. In practice each segment becomes a label stack entry, and the segment list becomes the label stack from MPLS. The active segment is the top of stack. The SR operations also translates into MPLS:

- PUSH becomes PUSH,
- NEXT becomes POP, and
- CONTINUE becomes SWAP with the same label, essentially NO-OP.

1.2 Related Work

1.2.1 Network Verification

Typically network verification tools are given some configuration (either control or data plane) and one or more properties to check.

FSR [17] is oriented towards the goal of verifying if proposed constraints on routing policy convergence, it uses RapidNet [18] as a distributed implementation, and the Yices[19] SMT solver to analyze safety results. FSR is only applicable for Border Gateway Protocol (BGP) and therefore it does not compare with our tool which is oriented towards MPLS

Batfish [20] takes the control plane and a specific scenario (e.g. a failure scenario) as input, and derives a data plane, which it uses to analyze for a broad range of properties. However it requires the network operator to enumerate all failure scenario combination, and their approach does not support MPLS.

1.2.2 Network Configuration Synthesis

In contrast to verification of existing configurations are synthesis of new ones, which often aims to be correct by construction. Tools that do this are referred to as synthesis tools, the network operator provides the tool with a specification of rules that the configuration must adhere to, and the tool then generates a

Features/Tools	Prex	NetKAT	HSA	VeriFlow	Anteater	SyNET
Protocol Support	(SR-)MPLS	OpenFlow	Agnostic	OpenFlow[a]	Agnostic	OSPF & BGP
Modeling Approach	Automata Theoretic	Algorithmic	Geometric	Trie-graphs	SAT problems	N/A
Complexity	Polynomial	PSPACE	$O(R1R2)^b$	NP-Complete	NP-Complete	N/A
Static	✓	✓	✓	χ	✓	✓
Reachability Queries	✓	✓	✓	✓	✓	χ
Forward Loop Queries	✓	✓	✓	✓	✓	χ
Reachability under Failure	✓	N/A	✓	N/A	χ	N/A
Unlimited Header Size	✓	N/A	χ	χ	N/A	N/A
Synthesis	χ	χ	χ	χ	χ	✓
Practical Performance	N/A	✓[27]	✓[c]	✓[d]	✓[e]	✓[26]
Waypointing	✓	✓	✓	✓	χ	✓
Language	Python & C	OCaml	Python & C	Python	C++ & Ruby	Python

[a]With the execption of packet transformation protocols like MPLS

[b]For reachability. $R1$ = Union of wildcard expressions, $R2$ = Number of transfer function rules. While not exponential this ends up being a very high polynomial scale, based on the results in Section 6

[c]Tests on Standford University's backbone Network showed a performance of 151 seconds for model generation with reachability queries averaging 13 seconds and loop queries averaging 18.6 seconds per port.

[d]Reachability queries performed in 75.39 ms on the same network which Header Space Analysis (HSA) performs them in 578.39 ms

[e]approximately 30 minutes on a network with 384 nodes

Table 3: Comparsion of related tools.

network configuration. Synthesis is a significantly different approach to the problem resulting in no meaningful comparisons to Prex.

Propane [21, 22] and Propane/AT [23] is a language and compiler for synthesizing BGP router configurations, based on a high level policy which they guarantee to implement under all possible failures. Centrally they are interested in keeping private traffic from leaking to non trusted parties.

NetComplete [24] aims to assist operators in modifying existing network-wide configurations, to accommodate changes in routing policies, their core selling point is that it allows "holes" in the configurations, which it can then "autocomplete". Central to their approach is counter-example guided inductive synthesis[25] which gives them significant speedups ($> 600\times$) relative to their competitor SyNET[26]. They do not support MPLS, but they plan to add support for it.

SyNET[26] aims to provide a provide a functioning network configuration for networks using multiple interacting protocols, namely Open Shortest Path First (OSPF) and BGP. To create a configuration SyNET takes a network specification, a network topology, and a set of requirements. SyNET[26] was created under the assumption that most networks are not Software Defined Network (SDN) and as such requires each router to be configured individually. This is reflected in the output where a configuration for each router is given, both the input and output is given in stratified Datalog.

1.3 Verification Tools

The surge of interest in SDNs have brought with it the development of various SDN verification tools. Despite all the tools falling into the same category, SDN verification tools, they exhibit some significant differences. NetKAT, VeriFlow, HSA, and Anteater are all such verification tools [28, 27, 29, 30]. The tool developed in this paper is no different, sharing in the goals of a SDN verification tool, but doing so with a different approach. In Table 3 we compare the central

points in five verification tools, with our own tool: Prex.

NetKAT focuses on static verification of the network configuration and supports checking for failures in terms of reachability and forwarding loops, with a support for waypointing. NetKAT[27] sets itself apart from our, and other tools, particularly in its approach to modeling and expressing the network. NetKAT is based on Kleene Algebra with Tests (KAT), an algebraic system based on Kleene Algebra (KA), the algebra used for regular expressions[31, 32, 27]. This in conjunction with other mathematical concepts, leads to NetKAT expressing the network and its relations through what they call NetKAT expressions, rather than encode the topology and policy as a logical structure. NetKAT uses NetKAT expressions for both encoding of networks and to express queries. This is similar to our tool in that regular expressions are used as the basis for queries.

Header Space Analysis is similarly to NetKAT a static verification tool. As the name suggests, this tool is focused on utilizing the headers of packets for the verification. HSA[29], in terms of failures, covers reachability, forwarding loops, traffic isolation and leakage. Unlike NetKAT, HSA uses a logical modeling approach, generating a geometric model from the packet headers and the network configuration. In order to model this the headers are bounded, and their protocol-specific meanings are ignored such that the tool can be protocol agnostic. The tool developed in this paper removes the restriction on headers by being unbounded in size. This tool is similar in that it focuses on headers, and is a static verification tool, but the approach used is distinctly different from ours.

VeriFlow unlike both our and the other mentioned tools, VeriFlow[28] focuses on being able to detect bugs as they occur, if not prevent them all together. This tool is effectively added to the networks configuration and acting as a layer between the network and the SDN controller. VeriFlow works by verifying subsets of the network that would be effected by an update packet from the SDN controller. VeriFlow models data-plane information as boolean expressions and uses an SAT solver algorithm to check for failures. SAT is an NP-Complete problem, as such SAT solvers generally use heuristics and approximations in order to perform closer to polynomial time, as is the case for the SAT solvers used by VeriFlow and Anteater. Checking only subsets of the network at a time, lowers VeriFlow's influence on the network performance, [28] shows this to cause approximately 15 % increase in latency.

Anteater in terms of approach, Anteater[30] 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. In terms of goal, Anteater is more similar to HSA, in that it is a static tool used to analyze an already existing configuration, similar also to our tool. Anteater focusing mainly on algorithms to detect reachability, forwarding loops, and packet loss as invariants; and modeling packets to support multiple protocols. The remaining parts of the tool, code generation, SAT solving, linking, and optimization is left to off-the-shelf solutions. In [30] no theoretical complexity analysis is made, although a

number of tests are performed. The data implies that the SAT solver used is of quadratic complexity, i.e. it finds heuristics in quadratic time. However, the SAT solving part of ranges from being approximately 10 - 30 % of the complete runtime depending on the invariant being checked for. The results are on a single network with a particular topology and hierarchical network complexity, as a result the runtime may be specific to the single network, as the paper also mentions that Anteater's runtime depends on the network topology and complexity. Anteater is capable of finding multiple kinds of issues such as forwarding loops, packet loss, and inconsistencies in the Access Control List (ACL) rules. While no complexity outside of the SAT solving is provided, it is mentioned that Anteater took about half an hour to check static properties in a network of 384 nodes. In terms of approach there are no real similarities between Anteater and our tool.

While the tools have clear differences, they all serve the same purpose, reachability and forward loop detection of networks. Both a static and a more involved process is used, as well as logical and algebraic modeling, yet none of the tools mention link failures, or handling thereof. To the best of our knowledge, this is not something the aforementioned tools have concerned themselves with, and is the primary focus of the tool developed in this paper. The tools also mention little in regard to performance that is easily comparable, merely results on specific use cases. HSA and VeriFlow does share one such use case, leading to some comparison; according to [28] VeriFlow outperforms HSA, at least in the case of finding reachability problems on the specific use case of Standford University's backbone network.

1.4 Problem & Overview

This section provides the problem which we will aim to solves, as well as provide an overview of how our solution will work.

Problem 1 (Query Satisfiability Problem)**.** Given a network topology, the routing tables thereof, and a query, does there exists a trace through the network which satisfies the query?

The exact definitions of network topology, routing tables and query will be given later in the paper.

1.4.1 Overview of solution

The input to our method is a network topology and routing tables, either in the form of our XML format or data from a set of Juniper routers, and a query. Figure 4 gives an overview of the steps we take to determine if the query either is satisfied or not. In Section 2.1 on page 12 we introduce our MPLS network model, and related definitions. We then introduce a number of concepts regarding pushdown and non-deterministic finite automata.

Then we encode our network model as a Pushdown Automaton (PDA) such that we can query on it. The queries consist of four parts:

- A regular expression specifying the language of the initial header,
- A regular expression specifying the language of the path through the network,

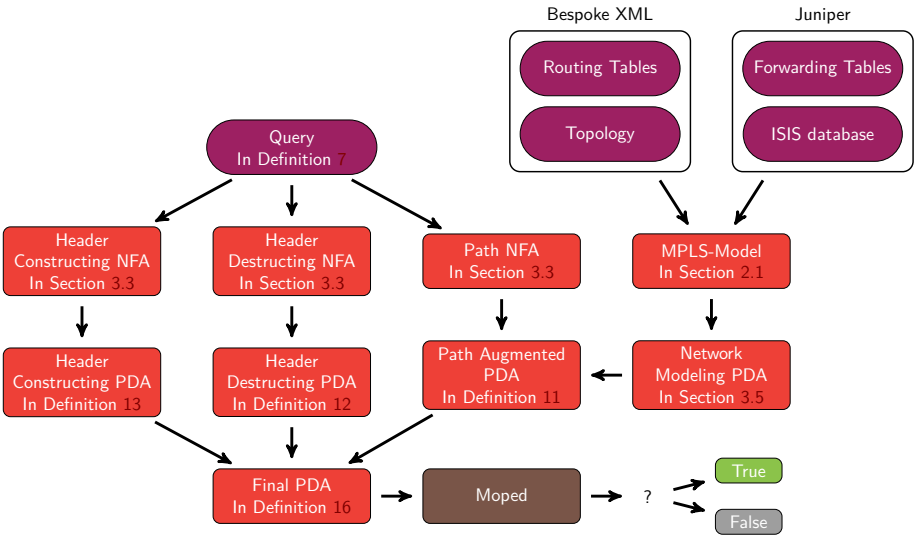


Figure 4: Flowchart of how we solve the problem.

- A regular expression specifying the language of the final header,
- The maximum number of failed links.

The initial header and final header regular expressions are each converted to first an Nondeterministic Finite Automaton (NFA) and then to a PDA. The path query is converted to an NFA, which is used to augmented the PDA constructed based on the network model. The three PDAs are combined into a single PDA which we give to a PDA reachability tool called Moped [33]. Moped will then either provide a trace through the pushdown which witnesses the query, or says that no such witness exist.

1.5 Our Contribution

This paper presents a tool which is capable of running reachability queries, written as regular expressions, on MPLS networks in failure scenarios. We base ourselves on a key insight from Schmid and Srba [34] which describe how to emulate an MPLS network using prefix rewrite systems, specifically PDA. While our model of MPLS networks are based on their we extend it to be closer to actual implementations of MPLS, specifically Junipers [35]. We do this by incorporating the ability to perform multiple operations based on the top of stack label, and non-determinism to support traffic engineering in routing.

We introduce a novel method for combining an NFA, made from the query, and PDA into a single PDA which then simulates them running in lockstep. This method is used to restrict the paths through the PDA which emulates the MPLS behavior. Furthermore our query can restrict the initial and final headers of a packet trace through the network. To increase the performance of our tool, we introduce an optimization we call “top of stack reduction”, which safely calculates which labels can be at the top of stack in a given state of the PDA, this greatly reduces the amount of transitions in the PDA.

Lastly we perform a case-study on a real world network provided by

NORDUnet, in addition to comparing ourselves with a well known tool in networking.

2 Formal Network Model

In this section we present our formal definitions of MPLS networks, routing tables, and network traces. These modify and extend the network definitions presented by Schmid and Srba [34] in order to reflect our tool and SR enabled networks. We will explain the differences from the approach of Schmid and Srba in Section 2.3.

2.1 Networks Model

Definition 1 (MPLS Operations)**.** Let L be a nonempty, finite set of labels used in the MPLS-headers of the network. Each hop in the MPLS network can then apply a number of the following operations to the label stack:

$$Op = \{swap(\ell) \mid \ell \in L\} \cup \{push(\ell) \mid \ell \in L\} \cup \{pop\}$$

Definition 2 (MPLS Network)**.** We formally define a MPLS network model as $N = (V, I, L, E, \tau)$ where

- V is a finite set of routers,
- I is the finite set of all interfaces in the network, there exists a finite set for each router $v \in V$: I_v, which contains the interfaces on said router, the union of all interfaces on each router in the network is exactly equal to the set of all interfaces $I = \bigcup_{v \in V} I_v$, and we assume that all interfaces are unique to a router: $\forall v, v' \in V, v \neq v' \implies I_v \cap I_{v'} = \emptyset$,
- $L = M \uplus M^{\perp} \uplus L^{IP}$ is the set of the label stack entries, where M is the set of MPLS labels, $M^{\perp} = \{\ell^{\perp} \mid \ell \in M\}$ is the set of MPLS labels, with the bottom of stack bit set, and L^{IP} is a set of labels which are used to set the initial stack, based on IP routing information.
- E is the set of links between interfaces, expressed as pairs of connected interfaces $E \subseteq I \times I$. Any interface can only be connected to one other interface, and any link $(out, in) \in E$ has a corresponding entry $(in, out) \in E$. Formally:
  - $(out, in) \in E \implies (in, out) \in E$,
  - $(out, in), (out', in) \in E \implies out = out'$, and
  - $(out, in), (out, in') \in E \implies in = in'$.
- $\tau : I \times L \to (2^{I \times Op^*})^*$ is the global routing table, the image of τ is a sequence of sets, the routing is a non-deterministic choice between the members of each set, and the order in the sequence determines which set to route from, the lowest index set in the sequence has the highest priority, if the output is the empty sequence, then the packet is dropped. An additional restriction is that for any input interface, the output interface much be in the same router, that is $\forall (out, ops) \in \bigcup_{O \in \tau(in,\ell)} O : out \in I_v$ if and only if $in \in I_v$.

Links in the network may fail, throughout the paper we use $F \subseteq E$ as the set of failed links. Moreover the interface in are said to be active, if and only if it is connected to another interface, and the link is not failed. Formally

$\exists in' \in I, (in, in') \in E \setminus F$. Note that we allow loopback interfaces, which are connected to themselves, $in \in I, (in, in) \in E$.

MPLS networks often tunnel traffic containing some underlying header. We model these underlying, non MPLS, headers by the $\ell \in L^{IP}$ labels. Additionally, MPLS labels have a stacking bit, which we model by enforcing that only certain labels $\ell \in M^{\perp}$ can be at the bottom of the MPLS stack. A common model convention is to have $M^{\perp} = \{\ell^{\perp} \mid \ell \in M\}$. The structure of a valid header in our model is illustrated in Fig. 5, where a label is an element on the set at the corresponding level.

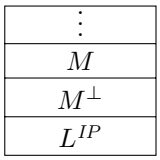


Figure 5: The structure of a valid header.

Definition 3 (Valid Headers)**.** We define a set of valid headers $H \subseteq L^*$. A valid header is either a member of L^{IP} or it consists of a member of L^{IP} as the bottom of the stack, then a single member of $M^{\perp}$ followed by 0 to more members of M:

$$H = \{\ell \mid \ell \in L^{IP}\} \cup \{\alpha_M \ell_{M^{\perp}} \ell_{L^{IP}} \mid \alpha_M \in M^*, \ell_{M^{\perp}} \in M^{\perp}, \ell_{L^{IP}} \in L^{IP}\}$$

The operations defined in Definition 1 operate on the header switching out labels with each other, pushing new labels, and removing old labels from the top of the stack. Throughout all of these operations, we want to ensure that our header remains valid $h \in H$.

Definition 4 (Header Rewrite Function)**.** We define the partial header rewrite function, $\mathcal{H} : H \times Op^* \hookrightarrow H$, for all valid packet header transformations for a given network $N = (V, I, L, E, \tau)$, $ops, ops' \in Op^*$, $\ell \in L$, and $h \in H$ as:

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

To give a concrete example, consider the alphabet, $L = \{10, 20, 30, 40\}$, where $M = \emptyset$, $M^{\perp} = \{20, 30\}$, and $L^{IP} = \{10, 40\}$, and the header $20 \circ 10 \in H$. Then $\mathcal{H}(20 \circ 10, pop \circ swap(40) \circ push(30)) = 30 \circ 40$. This calculation is also shown in Table 4 where each row corrosponds to a call to the header rewrite function.

Header	Operations remaining
$20 \circ 10$	$pop \circ swap(40) \circ push(30)$
10	$swap(40) \circ push(30)$
40	$push(30)$
$30 \circ 40$	ϵ

Table 4: Example of MPLS Operations applied to a header

Fact 1. The set of valid headers H is *partially* closed under the header rewrite function $\mathcal{H}$. That is, applying the header rewrite function to a valid header $h \in H$ and some operation chain $ops \in Op^*$ will map to some $h' \in H$ or be undefined:

$$\forall h \in H, \forall ops \in Op^* : \mathcal{H}(h, ops) \in H$$

As such, the header rewrite function preserves the validity of a header.

2.2 Traces

Informally a trace through the network is the path a packet, starting at some interface on a router with some header, takes through the network, before arriving at some destination interface of a router with a different header.

Let the set of interfaces in a traffic engineering group O be defined as $I(O) = \{in \mid (in, ops) \in O\}$, A traffic engineering group O is then said to be active if and only if for all $in \in I(O)$ and $in' \in I$, $(in, in') \notin F$.

We can now define the set of active rules over a sequence of traffic engineering groups, as mapped to by τ, as $\mathcal{A}(R) = \{(in, ops) \in O_j \mid \forall in' : (in, in') \notin F\}$, where $R = O_0 O_1 \dots O_n$, and j is the lowest index where O_j is active.

Definition 5 (A network trace)**.** We define a trace through a network, $N = (V, I, L, E, \tau)$, with the set of failed links $F \subseteq E$, given a packet header of h_1 starting at in_1, as a finite sequence of pairs:

$$\begin{gathered}(in_1, h_1),\\ (in_2, h_2),\\ \vdots\\ (in_n, h_n)\end{gathered}$$

where $(in_1, ops) \in \bigcup_{O \in \tau(in, \ell)} O$ for $\ell h' = h_0$ and some $in \in I$ let $h_1 = \mathcal{H}(h_0, ops)$. Furthermore, for all $1 \leq i \leq n-1$, let $(in_{i+1}, ops) \in \mathcal{A}(\tau(in_i, \ell))$ where $\ell h' = h_i$, and $h_{i+1} = \mathcal{H}(h_i, ops)$.

Additionally, we define the set of all such valid traces in a given network $N = (V, I, L, E, \tau)$ with the set of failed links F as the set of valid traces called $\mathcal{T}_N^F$.

2.3 Novelty in Network Model

As previously noted, we base our MPLS network model on one presented by Schmid and Srba in [34], however there are several modifications and extensions

performed to it. Our model allows multiple links between the same routers, such that they can fail on an individual basis. Our model supports multiple operations performed by the routing function on the same incoming header, based on the top of stack label. Our header rewrite function ensures that only valid formed MPLS label stacks are permitted. Our routing table function is non-deterministic, which allows TE behavior like load-balancing to occur.

All these changes are motivated by real world MPLS networks, in particular by Juniper[35].

2.4 Query Language

In this section we present our query language syntax based on regular expressions.

The queries consist of four parts:

- A regular expression specifying the language of the initial header,
- A regular expression specifying the language of the path through the network,
- A regular expression specifying the language of the final header,
- The maximum number of failed links.

Definition 6 (Regular Expression for Querying)**.** A regular expression for querying in our language is defined over some alphabet Σ and the symbols it contains $s \in \Sigma$, we write the class of regular expressions over an alphabet Σ as: $Reg(\Sigma)$.

$$a ::= s \mid . \mid [s...s_n] \mid [\hat{}s...s_n] \mid a^* \mid a^+ \mid a? \mid a_1a_2 \mid a_1\,'|'\,a_2$$

where the symbols represent the following:

s	is a symbol in Σ,
$.$	is a wildcard,
$[s...s_n]$	is a set of symbols belonging to the alphabet Σ,
$[\hat{}s...s_n]$	is the negated set of symbols belonging to the alphabet Σ,
a^*	is an arbitrary number of a,
a^+	is at least one a,
$a?$	is 0 or 1 a,
a_1a_2	is a concatenation of a_1 and a_2, and
$a_1 \mid a_2$	is a_1 or a_2.

Definition 7 (Query)**.** A query for some network $N = (V, I, L, E, \tau)$ has the following syntax:

$$'<'\,a\,'>'\,b\,'<'\,c\,'>'k$$

where $a, c \in Reg(L)$ and $b \in Reg(V)$ are regular expressions as defined in Definition 6 over the alphabets L and V respectively, and k is the maximum number of link failures.

Definition 8 (Trace Satisfying Query)**.** A trace $t = (in_1, h_1), (in_2, h_2), \ldots, (in_n, h_n)$ with F as the set of failed links in the network $N = (V, I, L, E, \tau)$ satisfies a query $q =< a > b < c > k$, and $|F| \leq k$, if and only if:

$$h_1 \in L(a),$$
$$h_n \in L(c),$$
$$in_1 in_2 \ldots in_n \in L(b).$$

From the the following problem naturally follows:

Problem 2 (Query Satisfiability Problem)**.** Given a network $N = (V, I, L, E, \tau)$ and a query q, is there a valid trace in the network $t \in \mathcal{T}_N^F$ where $|F| \leq k$ that satisfies q?

2.5 Network Example

To demonstrate our network model, and related functions, we provide the following example network, $N = (V, I, L, E, \tau)$, where

- $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$,
- $I = \{v_1^{in_1}, v_1^{v_2}, v_1^{v_3}, v_2^{in_2}, v_2^{v_1}, v_2^{v_4}, v_3^{v_1}, v_3^{v_4}, v_3^{v_5}, v_4^{v_2}, v_4^{v_3}, v_4^{v_6}, v_5^{v_3}, v_5^{v_6}, v_5^{out_1}\} \cup \{v_6^{v_4}, v_6^{out_2}\}$,
- $L = M \uplus M^{\perp} \uplus L^{IP}$ where
  - $M^{\perp} = \{10, 11, 20, 21, 30, 31, 32, 40, 41\}$,
  - $M = \{101, 102, 201, 202, 211, 212, 221, 222\}$, and
  - $L^{IP} = \{ip_{out_1}, ip_{out_2}\}$,
- $E \supseteq \{(v_1^{v_2}, v_2^{v_1}), (v_1^{v_3}, v_3^{v_1}), (v_2^{v_4}, v_4^{v_2}), (v_3^{v_4}, v_4^{v_3}), (v_3^{v_5}, v_5^{v_3}), (v_4^{v_6}, v_6^{v_4})\} \cup \{(v_5^{v_6}, v_6^{v_5})\}$, with additional tuples to satisfy Definition 2,
- τ is defined using the table in Table 5. Instead of the sequence returned by τ we give each rule a priority, lowest number indicating the highest priority, such that the number in the priority column is equal to the index in the sequence returned by τ, all priorities over 1 are fast failover rules.

The network is shown in Fig. 6 on page 18.

In the network we present four example LSPs , going from in_1 and in_2 to out_1 and out_2. The LSP from in_1 to out_1 could be witnessed by a packet with the following trace:

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

This trace follows rules of lines: 2, 10, and 22, in Table 5, the last tuple in the trace is the result after line 22.

The LSP from in_1 to out_2 takes the path: v_1, v_3, v_4, v_6. From $v_2^{in_2}$ to $v_5^{out_1}$ there is: v_2, v_4, v_3, v_5, and from $v_2^{in_2}$ to $v_6^{out_2}$ there is: v_2, v_4, v_6.

In the network two links are protected: (v_1, v_3) and (v_4, v_6). To protect these, for each LSP going through the link needs a backup LSP. For the case of (v_1, v_3) this path is v_1, v_2, v_4, v_3, and for (v_4, v_6) the path is v_4, v_3, v_5, v_6. In the routing table these are shown as having a lower priority than the normal LSPs. Moreover they utilize multiple operations when the FRR occurs, one is to push the backup LSP and one is to perform the action which would normally happen.

Consider the following query:

$$q_1 = < . > v_1 \ .^* \ v_6 < . > 2$$

Which means: Does there exist a trace through N, where starting at v_1 with a single label on the stack, through an arbitrary number of arbitrary routers in the network, to v_6 where only one label is on the stack, with a maximum of two link failures. The trace:

$$(v_1^{in_1}, ip_{out_2}), (v_3^{v_1}, 20 \circ ip_{out_2}), (v_4^{v_3}, 21 \circ ip_{out_2}), (v_6^{v_4}, 22 \circ ip_{out_2}), (v_6^{out_2}, ip_{out_2}).$$

with $F = \emptyset$ satisfies q_1, since all the conditions in Definition 8 are met by the trace. The network is shown in Fig. 6, with the trace satisfying q_1 shown as the blue path.

3 Formal Language Preliminaries

In this section we introduce the formal language preliminaries required for our approach.

3.1 Automata

This work is primarily interested in the two classes of automata, NFAs and PDAs. Since they will be important for the understanding, we will spend some space defining them.

Definition 9 (Non-deterministic Finite Automata)**.** A Nondeterministic Finite Automaton (NFA) is a 5-tuple $N = (S, \Sigma, \delta, s_0, s_f)$, where:

- S is a finite set of locations,
- 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 location, and
- $s_f \in S$ is the accepting location.

The computation of an NFA can be viewed as a series of configurations $(s, w) \in C(N) = S \times \Sigma^*$, where s is the current location and w is the remaining input. As the NFA computes it will follow transitions to different locations, while removing the symbol from the input. Thereby entering a new configuration $(s', w') \in C(N)$.

We introduce the infix arrow notation $\to_\delta \subseteq C(N) \times C(N)$ to symbolize emergent behaviour of the transition function δ as follows:

$$(s, w) \to_\delta (s', w) \text{ if } s' \in \delta(s, \epsilon)$$
$$(s, aw) \to_\delta (s', w) \text{ if } s' \in \delta(s, a)$$

for any $w \in \Sigma^*$ and $a \in \Sigma$. We will denote the transitive reflexive closure of the infix arrow as $\to_\delta^*$.

A string from the alphabet $w \in \Sigma^*$ is accepted by N if and only if there exists a series of configurations going from the initial to the final location, while reading exactly w. Formally $(s_0, w) \to_\delta^* (s_f, \epsilon)$. The set of all such strings w accepted by N is the language of N, formally: $L(N) = \{w \in \Sigma^* \mid (s_0, w) \to_\delta^* (s_f, \epsilon)\}$.

3.1.1 NFA Example

We will examine an NFA $N = (S, \Sigma, \delta, s_0, s_f)$ defined by the following components:

- $S = \{1, 2, 3, 4, 5, 6\}$,
- $\Sigma = \{q_0, q_1, q_2, q_3, q_5, q_f\}$,
- δ is defined by the graph in Fig. 7 where a transition from $s \in S$ to $s' \in S$ annotated by $a \in \Sigma$ means that $s' \in \delta(s, a)$,
- $s_0 = 1$,
- $s_f = 6$.

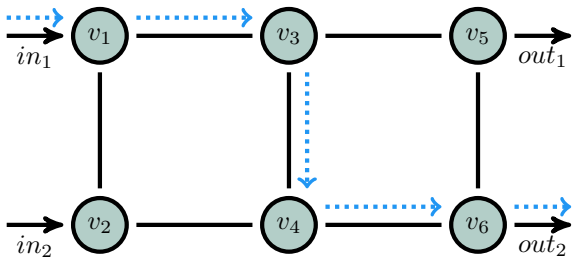


Figure 6: N_{exa} with the four LSPs shown.

1	Router	In I_v	Label	Prio	Out I_v	Operation
2	v_1	$v_1^{in_1}$	ip_{out_1}	1	$v_1^{v_3}$	push(10)
3		$v_1^{in_1}$	ip_{out_2}	1	$v_1^{v_3}$	push(20)
4		$v_1^{in_1}$	ip_{out_1}	2	$v_1^{v_2}$	push(10) ∘ push(101)
5		$v_1^{in_1}$	ip_{out_2}	2	$v_1^{v_2}$	push(20) ∘ push(201)
6	v_2	$v_2^{in_2}$	ip_{out_1}	1	$v_2^{v_4}$	push(30)
7		$v_2^{in_2}$	ip_{out_2}	1	$v_2^{v_4}$	push(40)
8		$v_2^{v_1}$	101	1	$v_2^{v_4}$	swap(102)
9		$v_2^{v_1}$	201	1	$v_2^{v_4}$	swap(202)
10	v_3	$v_3^{v_1}$	10	1	$v_3^{v_5}$	swap(11)
11		$v_3^{v_1}$	20	1	$v_3^{v_4}$	swap(21)
12		$v_3^{v_4}$	31	1	$v_3^{v_5}$	swap(32)
13		$v_3^{v_4}$	211	1	$v_3^{v_5}$	swap(212)
14		$v_3^{v_4}$	221	1	$v_3^{v_5}$	swap(222)
15	v_4	$v_4^{v_3}$	21	1	$v_4^{v_6}$	swap(22)
16		$v_4^{v_2}$	30	1	$v_4^{v_3}$	swap(31)
17		$v_4^{v_2}$	40	1	$v_4^{v_6}$	swap(41)
18		$v_4^{v_2}$	102	1	$v_4^{v_3}$	pop
19		$v_4^{v_2}$	202	1	$v_4^{v_3}$	pop
20		$v_4^{v_3}$	21	2	$v_4^{v_3}$	swap(22) ∘ push(211)
21		$v_4^{v_2}$	40	2	$v_4^{v_3}$	swap(41) ∘ push(221)
22	v_5	$v_5^{v_3}$	11	1	$v_5^{out_1}$	pop
23		$v_5^{v_3}$	31	1	$v_5^{out_1}$	pop
24		$v_5^{v_3}$	222	1	$v_5^{v_6}$	pop
25		$v_5^{v_3}$	212	1	$v_5^{v_6}$	pop
26	v_6	$v_6^{v_4}$	22	1	$v_6^{out_2}$	pop
27		$v_6^{v_4}$	41	1	$v_6^{out_2}$	pop

Table 5: Routing table for N_{exa} with (v_1, v_3) and (v_4, v_6) protected.

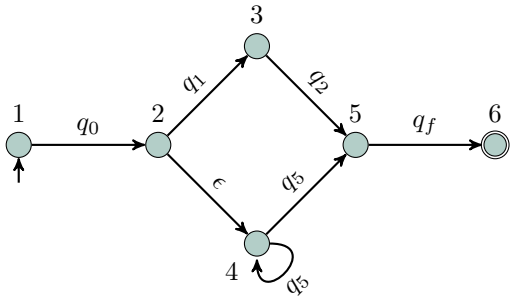


Figure 7: A simple NFA.

To provide an example of how a computation in an NFA works, consider the following example. Given $q_0q_5q_5q_f$ as input to N, we can get the following transitions:

$$(1, q_0q_5q_5q_f) \rightarrow_\delta (2, q_5q_5q_f) \rightarrow_\delta (4, q_5q_5q_f) \rightarrow_\delta (4, q_5q_f) \rightarrow_\delta (5, q_f) \rightarrow_\delta (6, \epsilon).$$

Since 6 is the final location of N, the string $q_0q_5q_5q_f$ is accepted, therefore $q_0q_5q_5q_f \in L(N)$.

An MPLS network maps very closely to PDA. A PDA has, much like the NFA a set of states, with transitions between them. The pushdowns examined in this paper do not have an input tape, but they will have the pushdowns defining feature, a stack to which the pushdown can read and write. The stack of a pushdown closely mirrors the stack of an MPLS network.

Definition 10 (Pushdown Automata)**.** A PDA is a 5-tuple $P = (Q, \Gamma, \lambda, q_0, q_f)$ where:

- Q is a finite set of locations,
- Γ is a finite stack alphabet,
- $\lambda : Q \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$ is the transition function. Where all sets in the co-domain are finite,
- $q_0 \in Q$ is the initial location, and
- $q_f \in Q$ is the final location.

A computation of a PDA can, like that of an NFA, be viewed as a series of configurations, expressed as 2-tuples $(q, h) \in C(P) = Q \times \Gamma^*$. The main difference being that where the NFA is only allowed to read from the symbol at the start of a word, the PDA can read from, push to, and swap from the top of its stack, essentially rewriting the prefix of the word in the stack.

We define the infix arrow notation $\rightarrow_\lambda \subseteq C(P) \times C(P)$ to, like that of the NFA, describe emergent behaviour of the transition function λ as follows: $(q, \ell h) \rightarrow_\lambda (q', \alpha h)$ if $(q', \alpha) \in \lambda(q, \ell)$, for all $\ell \in \Gamma$ and $h \in \Gamma^*$. We will denote the transitive reflexive closure of the infix arrow as $\rightarrow_\lambda^*$.

Notice that λ always takes exactly one symbol from the stack. Throughout this paper, $\perp$ will be used as a *bottom of stack* symbol, signifying the stack being empty. Since a pushdown removing the bottom of stack symbol would have no symbol to map with λ, it would effectively halt, we will not be popping the bottom of stack symbol.

3.1.2 PDA example

As an example consider the PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_f\}$,
- $\Gamma = \{10, 11, 20, \bot\}$,
- λ is defined by the figure Fig. 8, an edge from location q to q' annotated with $\ell \to \alpha$ means that $(q', \alpha) \in \lambda(q, \ell)$, edges without any annotation are rules of the form $(q', \ell) \in \lambda(q, \ell)$ for all applicable $\ell \in \Gamma$,
- $q_0 = q_0$, and
- $q_f = q_f$.

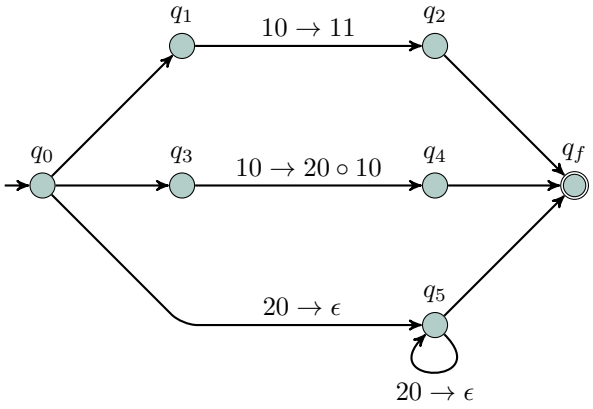


Figure 8: An example PDA

Consider the initial configuration $(q_0, 10 \circ \bot)$ which can reach q_f in two ways:

- $(q_0, 10 \circ \bot) \to_\lambda (q_1, 10 \circ \bot) \to_\lambda (q_2, 11 \circ \bot) \to_\lambda (q_f, 11 \circ \bot)$, and
- $(q_0, 10 \circ \bot) \to_\lambda (q_3, 10 \circ \bot) \to_\lambda (q_4, 20 \circ 10 \circ \bot) \to_\lambda (q_f, 20 \circ 10 \circ \bot)$.

The initial configuration $(q_0, 20 \circ 20 \circ \bot)$ can also reach q_f in two ways:

- $(q_0, 20 \circ 20 \circ \bot) \to_\lambda (q_5, 20 \circ \bot) \to_\lambda (q_f, 20 \circ \bot)$, and
- $(q_0, 20 \circ 20 \circ \bot) \to_\lambda (q_5, 20 \circ \bot) \to_\lambda (q_5, \bot) \to_\lambda (q_f, \bot)$.

3.2 Augmented Pushdown Construction

In this section we define our method of restricting the traces through a PDA to those accepted by an accompanying NFA.

Definition 11 (Augmented Pushdown)**.** Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and an NFA $N = (S, \Sigma, \delta, s_0, s_f)$ where $\Sigma \subseteq Q$, we can construct the pushdown $P' = (Q', \Gamma', \lambda', q_0', q_f')$ that computes the two automata in lockstep. This new pushdown P' will progress the original pushdown P iff N can read the destination location, while progressing N iff the transition is taken. Automaton N can therefore be thought of as a query on the path of P. The elements of the new augmented pushdown P' is defined as follows:

- $Q' = (Q \times S) \cup (\{start\} \times S)$ is the finite set of locations, with each either representing the current location of both automata, or the unique start location,
- $\Gamma' = \Gamma$ is the finite stack alphabet,
- $\lambda' : Q' \times \Gamma' \to 2^{Q' \times \Gamma'^*}$ is the transition function,

- $q'_0 = (start, s_0)$ is the initial location, and
- $q'_f = (q_f, s_f)$ is the final location.

where $start \notin Q$ is a new location acting as the initial location in P and allowing N to match the original initial location.

The transition function λ' drives the computation according to both original transition functions λ and δ as follows:

a) P' contains the rule:

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

for every $(q, s), (q, s') \in Q'$ and $\ell \in \Gamma'$ such that $s' \in \delta(s, \epsilon)$

b) P' contains the rule:

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

for every $(q, s), (q', s') \in Q'$, $\ell \in \Gamma'$, and $\alpha \in \Gamma'^*$ such that $s' \in \delta(s, q')$ and $(q', \alpha) \in \lambda(q, \ell)$

c) P' contains the rule:

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

for every $s, s' \in S$ and $\ell \in \Gamma'$ such that $s' \in \delta(s, q_0)$

Rule a) encodes N taking an epsilon transition, which neither affects the configuration of P, nor the stack of P'. rule b) is the main computational part. It encodes that if N has a transition that reads q' from s to s', and P has a transition from q to q' that rewrites the single symbol ℓ into the prefix α. Then P' must have a transition that progress both and modifies the stack. Analogous to rule b), rule c) starts off the computation by matching the original initial location in P.

Theorem 1. Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$, a NFA $N = (S, Q, \delta, s_0, s_f)$, and an initial header $h_0 \in \Gamma^*$, there exists a path through the pushdown $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_{n-1}, h_{n-1}) \to_\lambda (q_f, h_n)$ such that $q_0 q_1 \dots q_{n-1} q_f \in L(N)$ if and only if:

$$((start, s_0), h_0) \to^*_{\lambda'} ((q_f, s_f), h_n).$$

Proof. To prove Theorem 1 we will prove the stronger claim that $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_n, h_n)$ and $(s_0, q_0 q_1 \dots q_n) \to^*_\delta (s_n, \epsilon)$ if and only if

$$((start, s_0), h_0) \to^*_{\lambda'} ((q_n, s_n), h_n).$$

Theorem 1 then naturally follows.

Assume a trace $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_n, h_n)$ in P such that $(s_0, q_0 q_1 \dots q_n) \to^*_\delta (s', \epsilon)$. We want to show that there exists a corresponding trace $((start, s_0), h_0) \to^*_{\lambda'} ((q_n, s'), h_n)$ in P'.

Base Case ($n = 0$): By at most $|S|$ applications of rule a) followed by a single application of rule c) we can get the trace $((start, s_0), h_0) \to_{\lambda'} ((start, s_1), h_0) \to_{\lambda'} \cdots \to_{\lambda'} ((start, s_j), h_0) \to_{\lambda'} ((q_0, s_{j+1}), h_0)$. By then, another at most $|S|$ applications of rule a) can yield $((q_0, s_{j+1}), h_0) \to_{\lambda'} ((q_0, s_{j+2}), h_0) \to_{\lambda'} \cdots \to_{\lambda'} ((q_0, s'), h_0)$.

Inductive Step: Assume a trace in P $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_n, h_n)$ such that $(s_0, q_0 q_1 \dots q_n) \to_\delta^* (s', \epsilon)$ by the induction hypothesis P' will have a trace $((start, s_0), h_0) \to_{\lambda'}^* ((q_n, s'), h_n)$. Now assume another step in P, $(q_n, h_n) \to_\lambda (q_{n+1}, h_{n+1})$ such that $(s', q_{n+1}) \to_\delta^* (s'', \epsilon)$. We can capture this step by applying rule a) at most $|S|$ times followed by rule b) yielding the trace $((q_n, s'), h_n) \to_{\lambda'}^* ((q_{n+1}, s''), h_{n+1})$.

Contrapositively, Assume a trace in P':

$$\begin{aligned}
&((start, s_0), h_0) \to_{\lambda'} ((start, s_0^1), h_0) \to_{\lambda'} \cdots \to_{\lambda'} ((start, s_0^{l_0}), h_0) \\
&\quad \to_{\lambda'} ((q_0, s_1), h_0) \to_{\lambda'} ((q_0, s_1^1), h_0) \to_{\lambda'} \cdots \to_{\lambda'} ((q_0, s_1^{l_1}), h_0) \\
&\quad \to_{\lambda'} ((q_1, s_2), h_1) \to_{\lambda'} ((q_1, s_2^1), h_1) \to_{\lambda'} \cdots \to_{\lambda'} ((q_1, s_2^{l_2}), h_1) \\
&\qquad\qquad \vdots \\
&\to_{\lambda'} ((q_n, s_{n+1}), h_n) \to_{\lambda'} ((q_n, s_{n+1}^1), h_n) \to_{\lambda'} \cdots \to_{\lambda'} ((q_n, s_{n+1}^{l_{n+1}}), h_n)
\end{aligned} \tag{1}$$

We want to show that there exists a corresponding trace $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_n, h_n)$ in P such that $(s_0, q_0 q_1 \dots q_n) \to_\delta^* (s_{n+1}^{l_{n+1}}, \epsilon)$.

Base Case ($n = 0$): The sub-trace $((start, s_0), h_0) \to_{\lambda'}^* ((start, s_0^{l_0}), h_0)$ can only exist from rule a), which implies $(s_0, w) \to_\delta^* (s_0^{l_0}, w)$. The single step $((start, s_0^{l_0}), h_0) \to_{\lambda'} ((q_0, s_1), h_0)$ can only come from rule c), which implies that $(s_0^{l_0}, q_0 w') \to_\delta (s_1, w')$. Finally, rule a) is again required for the last sub-trace, implying $(s_1, w') \to_\delta^* (s_1^{l_1}, w')$.

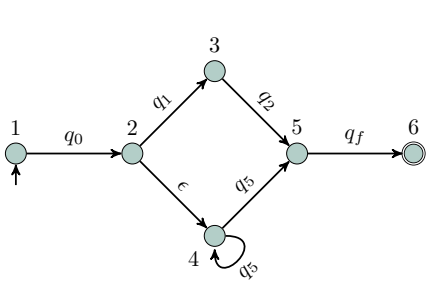
Inductive Case: Assume a trace in P' like that in Eq. (1). By the induction hypothesis, there exists a corresponding trace $(q_0, h_0) \to_\lambda (q_1, h_1) \to_\lambda \cdots \to_\lambda (q_n, h_n)$ in P such that $(s_0, q_0 q_1 \dots q_n w) \to_\delta^* (s_{n+1}^{l_{n+1}}, w)$. Now assume a next step in P', the trace $((q_n, s_{n+1}^{l_{n+1}}), h_n) \to_{\lambda'} ((q_{n+1}, s_{n+2}), h_{n+1}) \to_{\lambda'} ((q_{n+1}, s_{n+2}^1), h_{n+1}) \to_\lambda \cdots ((q_{n+1}, s_{n+2}^{l_{n+2}}), h_{n+1})$. The first sub-step implies, through rule b), that $(q_n, h_n) \to_\lambda (q_{n+1}, h_{n+1})$ and $(s_{n+1}^{l_{n+1}}, q_{n+1} w') \to_\delta (s_{n+2}, w')$. Subsequently, the remaining steps, because of rule a), imply that $(s_{n+2}, w') \to_\delta (s_{n+2}^{l_{n+2}}, w')$.

□

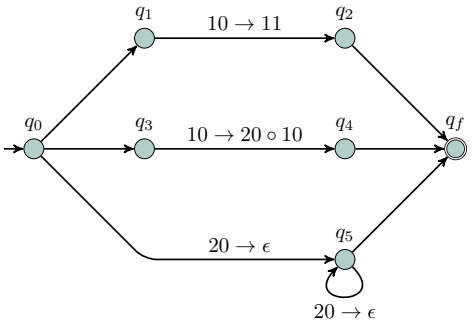
3.2.1 Augmented Pushdown Example

For an example we will examine an augmented pushdown constructed from the previously explored examples of an NFA in Section 3.1 and a PDA in Section 3.1. Observe that these fulfil the requirements to construct an augmented pushdown, namely $\Sigma \subseteq Q$. The elements of the constructed augmented pushdown $P' = (Q', \Gamma', \lambda', q_0', q_f')$ is as follows:

- $Q' = \left\{ \begin{matrix} (q_0,1), (q_1,1), \dots, (q_f,1), \\ (q_0,2), (q_1,2), \dots, (q_f,2), \\ \vdots \\ (q_0,6)(q_1,6), \dots, (q_f,6) \end{matrix} \right\} \cup \{(start,1), (start,2), \dots, (start,6)\}$
- $\Gamma' = \{10, 11, 20\}$,
- λ' is defined by Fig. 9,
- $q_0' = (start, 1)$, and
- $q_f' = (q_f, 6)$.



(a) NFA from NFA Example.



(b) PDA from PDA example.

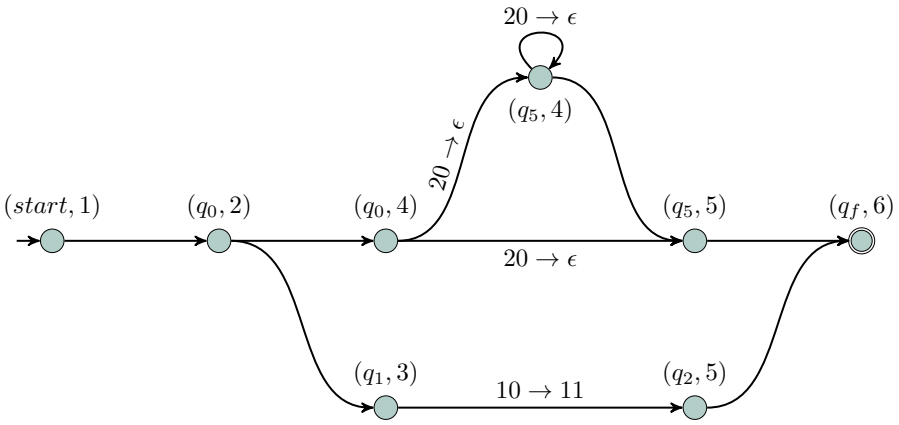


Figure 9: λ' constructed from Fig. 9a and Fig. 9b

In the original PDA P, from Section 3.1.2, there existed two computations starting at q_0 ending at the final location q_f with 10 as the starting stack, and ending with either 11 or $20 \circ 10$. In the augmented PDA P' there only exists one computation starting with the configuration $((start, 1), 10)$ that ends in the final location $(q_f, 6)$. This computation is as follows:

$$((start, 1), 10) \to_{\lambda'} ((q_0, 1), 10) \to_{\lambda'} ((q_0, 2), 10) \to_{\lambda'} ((q_1, 3), 10) \to_{\lambda'} ((q_2, 5), 11) \to_{\lambda'} ((q_f, 6), 11).$$

This trace corresponds to the following trace through P:

$$(q_0, 10) \to_{\lambda} (q_1, 10) \to_{\lambda} (q_2, 11) \to_{\lambda} (q_f, 11).$$

In lock step with the following trace through the NFA N from Section 3.1.1:

$$(1, q_0q_1q_2q_f) \to_{\delta} (2, q_1q_2q_f) \to_{\delta} (3, q_2q_f) \to_{\delta} (5, q_f) \to_{\delta} (6, \epsilon).$$

3.3 Regex to NFA

To compactly describe an NFA we will use regular expressions.

A regular expression, R, defines a specific search pattern, all strings included in said pattern creates the language $L(R)$[32]. From Kleene's theorem [32] we know that the same language can also be defined as a language recognized by an automaton. Additionally, from Thompson's Construction we know that it is possible to create an NFA from a regular expression, the runtime for which is linear in the size of the regular expression[36].

Theorem 2. [36] Given a regular expression R we can, in linear time, construct an equivalent NFA $N = (S, \Sigma, \delta, s_0, s_f)$ such that:

$$L(N) = L(R).$$

The reverse of a string w, in this paper denoted w^R, is defined by the property that if $w = w_0w_1 \dots w_n$ then $w^R = w_nw_{n-1} \dots w_0$. The reverse of a language L can also be defined as $L^R = \{w^R \mid w \in L\}$. We can construct the reverse NFA N^R in linear time.

Theorem 3. [37] Given an NFA $N = (S, \Sigma, \delta, s_0, s_f)$, we can construct an NFA N^R recognizing the reverse language $L^R(N)$.

We extend the $\in$ notation to cover symbols in strings. As such, $a \in w$ will mean $a \in \{w_i \mid 0 \leq i \leq n\}$, where $w = w_0w_1 \dots w_n$.

3.4 Converting an NFA to a PDA

We will now describe our method of converting an NFA into a PDA that either reads a string $h \in L(N)$ from the stack, or places one string $h \in L(N)$ on the stack.

Intuitively converting an NFA $N = (S, \Sigma, \delta, s_0, s_f)$ into a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ reading all strings $w \in L(N)$ from the stack, such that $(q_0, w\perp) \to_{\lambda} (q_f, \perp)$ is trivial. Such a PDA will simply remove a label when the NFA would read it and swap the top of stack label $\ell' \in \Gamma$ with itself when N has an epsilon transition.

Definition 12 (Destructing PDA)**.** Given an NFA $N = (Q, \Sigma, \delta, q_0, q_f)$, the corresponding destructing PDA $P_d = (Q, \Gamma, \lambda, q_0, q_f)$ is defined as follows: Notice that Q, q_0, and q_f are same in both P_c and N, leaving only Γ and the transition function λ in need of a new definition:

- $\Gamma = \Sigma \cup \{\bot\}$,
- λ is the transition function defined below,

a) P_d has the rule:

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

for every $q, q' \in Q$ and $\ell \in \Sigma$, such that $q' \in \delta(q, \ell)$.

b) P_d has the rule:

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

for every $q, q' \in Q$ and $\ell \in \Gamma$, such that $q' \in \delta(q, \epsilon)$.

Theorem 4. Given an NFA $N = (Q, \Sigma, \delta, q_0, q_f)$, the constructed PDA $P_d = (Q, \Gamma, \lambda, q_0, q_f)$, satisfies $(q_0, h) \to_\delta^* (q_f, \epsilon)$ if and only if

$$(q_0, h\bot) \to_\lambda^* (q_f, \bot).$$

Lemma 1. To prove Theorem 4 we will prove the stronger claim that that $(q_0, h) \to_\delta^n (q', h')$ if and only if $(q_0, h\bot) \to_\lambda^n (q', h'\bot)$. From there the above theorem naturally follows.

Proof. We will prove the above lemma by induction on the length, n, of the trace.

Assuming a trace $(q_0, h) \to_\delta^n (q', h')$ in N, we will show that there exists a trace $(q_0, h\bot) \to_\lambda^n (q', h'\bot)$ in P.

Base Case ($n = 0$): The base case is trivially true.

Inductive case: Assume a trace $(q_0, h) \to_\delta^n (q', h')$ in N, by the induction hypothesis there is a trace $(q_0, h\bot) \to_\lambda^n (q', h'\bot)$. Assume an extra step in N, $(q', h') \to_\delta (q'', h'')$. If $\ell h'' = h'$ this step can be captured by rule a) and the trace $(q', \ell h''\bot) \to_\lambda (q'', h''\bot)$. Alternatively, if $h' = h''$, the step can be captured by rule b) by the trace $(q', h''\bot) \to_\lambda (q'', h''\bot)$.

For the contraposition, Assume a trace $(q_0, h\bot) \to_\lambda^n (q', h'\bot)$ in P, we will show that there exists a trace $(q_0, h) \to_\delta^n (q', h')$ in N.

Base Case ($n = 0$): The base case is trivially true.

Inductive case: Assume a trace $(q_0, h\bot) \to_\lambda^n (q', h'\bot)$ in N, by the induction hypothesis there is a trace $(q_0, h) \to_\delta^n (q', h')$. Assume an extra step in N, $(q', h'\bot) \to_\lambda (q'', h''\bot)$. If $\ell h'' = h'$ this step can only come from rule a), in which case $(q', \ell h'') \to_\delta (q'', h'')$. Otherwise, if $h' = h''$, the step is a result of rule b), and $(q', h'') \to_\delta (q'', h'')$.

□

We are also interested in the opposite construction, taking an NFA N and creating a PDA P that can reach q_f with all stack words $h \in L(N)$. Intuitively, this is possible by thinking about the NFA writing to the input tape when it would otherwise read, except that produces reversed strings. To counteract this reversal we first need to reverse the NFA, as discussed in Theorem 3. From this new *Writing Reversed* NFA construction of the PDA is similar to Definition 12.

Definition 13 (Constructing PDA)**.** Given an NFA $N = (Q, \Sigma, \delta, s_0, s_f)$, we define the *Constructing pushdown* $P_c = (Q, \Gamma, \lambda, q_0, q_f)$. The first step is to reverse N, yielding $N^R = (Q, \Sigma, \delta^R, q_0, q_f)$, we then construct P_c from N^R:

- $\Gamma = \Sigma \cup \{\bot\}$,
- λ is the transition function defined below,

Notice that Q, q_0, and q_f are the same in P_c and N^R. The direct construction yields the following the following definition of λ:

a) P_c has the rule:

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

for every $q, q' \in Q$, $\ell \in \Gamma$ and $\ell' \in \Sigma$, such that $q' \in \delta^R(q, \ell')$.

b) P_c has the rule:

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

for every $q, q' \in Q$ and $\ell \in \Gamma$, such that $q' \in \delta^R(q, \epsilon)$.

Informally, the first rule a) λ pushes the label ℓ' when δ reads l, rule b) λ does not modify the stack when δ reads ϵ.

Theorem 5. Given an NFA $N = (Q, \Sigma, \delta, q_0, q_f)$, the constructed Constructing PDA $P_c = (Q, \Gamma, \lambda, q_0, q_f)$ satisfied $(s_0, h) \to_\delta^* (s_f, \epsilon)$ if and only if

$$(q_0, \bot) \to_\lambda^* (q_f, h\bot).$$

The proof of Theorem 5 is similar to that of Theorem 4.

3.5 Encoding MPLS Reachability Into PDAs

We show how to encode the MPLS network model presented earlier, into an PDA, such that we can perform reachability analysis on it. Since the naive approach to computing the exact network reachability under failures, in which you enumerate all failure cases, grows exponentially in the number of failed links, we will instead show how to over-approximate the reachability.

3.5.1 Over-approximation

Intuitively we encode the network as follows:

- The location of the pushdown, with the exception of the initial and final locations q_0 and q_f, are in the form of triples: (out, ops), where $out \in I$ is the outgoing interface, $ops \in Op^*$ is the remaining operations.
- The stack of the pushdown will exactly be the MPLS label stack, and any stack in the pushdown will also be a valid header.

In order to support multiple operations internally in the routers, we include the pending operations, *ops*, as part of the locations. When $ops = \epsilon$ there are no remaining operations to perform on the packet, and the next transition will be to what is logically another router. We expect that there being only a single operation to be the most common occurrence, but multi-operation is crucial, for fast-failover for example.

We observe that an mpls interface can, according to the Definition 2 rule for E only be connected to one other interface. An outgoing interface *out*

can therefore uniquely identify the incoming interface in it is connected to, $(out, in) \in E$. For this reason, we do not include locations for incoming interface, but rather connect outgoing interfaces out to the ones the corresponding incoming interface in can send to.

There exists a finite set of possible operation sequences and subsequences for the τ for any network, this set is denoted as $\overline{Ops}$.

The following conversion rules were initially given by Schmid and Srba [34], but heavily adapted to the MPLS model presented inSection 2.1.

Given an MPLS network $N = (V, I, L, E, \tau)$ and a maximum number of link failures k such that $0 \leq k \leq |E|$, we define the pushdown system $P(N) = (Q, \Gamma, \lambda, q_0, q_f)$ as:

- $Q = \{(out, ops) \mid out \in I, ops \in \overline{Ops}\} \cup \{q_0, q_f\}$,
- $\Gamma = L$,
- λ is defined in terms of its rules below,
- $q_0 = q_0$, and
- $q_f = q_f$.

Recall that the routing function τ maps an interface and a label to a string of traffic engineering groups $\tau(in, \ell) = O_0 O_1 \dots O_n$, which are themselves sets of interface operation chain pairs $O_c = \{(in_c^1, ops_c^1), (in_c^2, ops_c^2), \dots, (in_c^{l_c}, ops_c^{l_c})\}$ for all $0 \leq c \leq n$. The failure aware routing function is defined as $\tau^k(in, \ell) = \bigcup_{j=0}^{i} O_j$, such that i the smallest index where:

$$\left| \begin{Bmatrix} in_1^1, in_1^2, \dots, in_1^{l_1}, \\ in_2^1, in_2^2, \dots, in_2^{l_2}, \\ \vdots \\ in_i^1, in_i^2, \dots, in_i^{l_i}, \end{Bmatrix} \right| > k$$

a) $P(N)$ contains the rule

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

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

b) $P(N)$ contains the rule

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

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

c) $P(N)$ contains the rule

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

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

d) $P(N)$ contains the rule

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

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

e) $P(N)$ contains the rule

$$((out, ops), \ell) \in \lambda(q_0, \ell)$$

for every $in \in I$, $\ell \in L$, and $(out, ops) \in \bigcup_{O \in \tau(in, \ell)} O$

f) $P(N)$ contains the rule

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

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

Rule a) connects the routers and also implements the routing tables. This is necessary in our construction since we have do not have locations in the PDA, which represent the incoming interfaces. Therefore the outgoing interfaces of one router needs to connect to the outgoing interfaces with the remaining operations of the other router. Here we use the minimum between the number of failover rules and function $\mathcal{C}$ to limit the number of failover rules to the ones allowed with the given k.

Rules b) to d) implements the header rewrite function from Definition 4. Specifically rule b) implements the swapping behavior, rule c) implements the pushing behavior, and rule d) implements the popping behavior. Moreover only the same cases as the header rewrite functions are added to the pushdown.

In rule e) we connect the initial location of the pushdown to every logical incoming interface of the pushdown i.e. everywhere a multi-operation begins. Lastly in rule f) we connect everywhere the *ops* part of the location is ϵ to q_f, the final location in the pushdown.

Theorem 6. Given an MPLS network model $N = (V, I, L, E, \tau)$ and a maximum number of link failures k, we can construct the over-approximating PDA $P(N)$ for k, such that there is a trace in the network $(in_0, h_0), (in_1, h_1), \ldots (in_n, h_n) \in \mathcal{T}_N^F$ and $|F| \leq k$ only if:

$$(q_0, h_0) \to_\lambda^* (q_f, h_n).$$

Proof. To prove Theorem 6 we will prove the stronger claim that a trace in the network $(in_0, h_0^{l_0}), (in_1, h_1^{l_1}), \ldots, (in_n, h_n^{l_n}) \in \mathcal{T}_N^F$ and $|F| \leq k$ will imply a trace in $P(N)$:

$$\begin{aligned}
&(q_0, h_0) \\
&\to_\lambda ((in_0, ops_0^0), h_0^0) \to_\lambda \\
&\quad ((in_0, ops_0^1), h_0^1) \to_\lambda \cdots \to_\lambda ((in_0, ops_0^{l_0-1}), h_0^{l_0-1}) \to_\lambda ((in_0, \epsilon), h_0^{l_0}) \\
&\to_\lambda ((in_1, ops_1^0), h_1^0) \to_\lambda \\
&\quad ((in_1, ops_1^1), h_1^1) \to_\lambda \cdots \to_\lambda ((in_1, ops_1^{l_1-1}), h_1^{l_1-1}) \to_\lambda (in_1, \epsilon, h_1^{l_1}) \\
&\qquad \vdots \\
&\to_\lambda ((in_1, ops_n^0), h_n^0) \to_\lambda \\
&\quad ((in_n, ops_n^1), h_n^1) \to_\lambda \cdots \to_\lambda ((in_n, ops_n^{l_n-1}), h_n^{l_n-1}) \to_\lambda (in_n, \epsilon, h_n^{l_n})
\end{aligned} \tag{2}$$

We will prove this by induction in n.

Base case ($n = 0$): The base case can be captured by an application of rule e) to simulate the initial application of the routing function from Definition 5, followed by l_0 applications of rules b) to d) to simulate the header rewrite function.

Inductive case: Assume a trace in the network $(in_0, h_0), (in_1, h_1), \ldots, (in_n, h_n)$, by the induction hypothesis there's a trace like Eq. (2). Now assume another tuple in the network trace (in_{n+1}, h_{n+1}), this step can be

captured by an application of rule a) simulating a link and the routing function, followed by l_{n+1} applications of rules b) to d) to simulate the header rewrite function.

To align the final location with that of Theorem 6, an application of rule f) do. □

There are several cases where the over-approximation contains behavior which is not in the MPLS model.

Too many fail-over rules taken : In the over-approximating PDA we allow there to be up-to k failures in each routing, this means that in a trace from it, there might have been taken a greater number of fast-failover rules than k, thus the PDA considers more links failed than k.

A previously considered failed interface is used : The over-approximating PDA does not save information about which links have previously failed. This can cause issues since if a fail-over rule is taken at some router v and then later in the packet routing v is visited again, then it can take an interface which was previously considered failed.

3.6 Under-approximation

The MPLS to PDA encoding shown above over-approximates the reachability of the N. Making it unsuitable for answering queries where a yes answer is desirable, since all yes queries are in fact a maybe.

We can modify the conversion process to instead under-approximate the reachability, turning a no answer into a maybe, but making a yes certain in the process. The intuition of the under-approximation method is that it's possible to add a variable i to all the states in the pushdown which encodes how many links have previously failed during this trace. As long as $i \leq |F|$, the problem, Section 3.5.1, of globally failing more links than allowed is resolved.

The problem of traveling through a link previously marked broken is more difficult. Perfectly eliminating the problem requires enumerating all failures, which won't be bounded by a polynomial. Instead, we restrict the trace by imposing that any trace $(in_0, h_0), (in_1, h_1), \ldots, (in_n, h_n)$ where $\exists 0 \leq i, j \leq n, v \in V : i \neq j$ and $in_i, in_j \in I_v$ is deemed inconclusive, a property which can quickly be checked on a trace.

The MPLS network model to PDA conversion can be modified such that it is an under-approximation, $P_u(N)$ which is usable in some scenarios.

Recall the failure aware routing function, τ. For under-approximation, we define a new routing function $\pi : I \times L \to \mathbb{Z} \times (2^{I \times Op^*})$. Where $\tau(in, \ell) = O_0 O_1 \ldots O_n$ such that $O_c = \{(in_c^0, ops_c^0), (in_c^1, ops_c^1), \ldots, (in_c^{l_c}, in_c^{l_c})\}$ for all $0 \leq n \leq k$, the routing function π maps an interface and a label to the set of tuples containing the number of failures and the TE group, let the number of failures at index i be

$$f_i = \left\{ \begin{array}{c} in_0^0, in_0^1, \ldots in_0^{l_0}, \\ in_1^0, in_1^1, \ldots in_1^{l_1}, \\ \vdots \\ in_{i-1}^0, in_{i-1}^1, \ldots in_{i-1}^{l_{i-1}}, \end{array} \right\}$$

then $\pi^k(in, \ell) = \{(f_i, O_i) \mid 0 \leq i \leq n, f_i \leq k\}$

The rules defined in Section 3.5.1 are modified as follows: rule a) is changed to

a) $P_u(N)$ contains the rule

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

for every $0 \leq i \leq k - j$, $\ell \in \Gamma$, and $(j, (out', ops)) \in \pi^k(in, \ell)$ where $in \in I$ such that $(in, out) \in E$.

Rule a) is the only rule responsible for increasing i, and also makes sure that only rules which are applicable, i.e. $i+j \leq k$ are added to the pushdown $P_u(N)$. rules b) to d) just forward the i part of the triple, rule e) will set $i = 0$, and rule f) for forwards for all $0 \leq i \leq k$.

Theorem 7. Given an MPLS network model $N = (V, I, L, E, \tau)$ and a maximum number of link failures k, we can construct the under-approximating PDA $P_u(N)$ for k, such that there is an acyclic trace in the network $(in_0, h_0), (in_1, h_1), \ldots, (in_n, h_n) \in \mathcal{T}_N^F$ where $|F| \leq k$ and $\nexists 0 \leq i, j \leq n, v \in V : i \neq j$ and $in_i, in_j \in I_v$ if:

$$(q_0, h_0) \rightarrow_\lambda^* (q_f, h_n)$$

4 Optimizations

After implementing the previously described methods, we hit several performance bottlenecks once we used non-trivial sized networks. In this section we document some of the optimization we have performed, to reduce the running time and memory requirement of our program.

4.1 Top of Stack reduction

The PDAs constructed in Sections 3.2, 3.4 and 3.5 contain many superfluous transitions. That is, transitions that read symbols from the stack that can not be at the top in a given location. Mathematically, these transitions will not pose a problem, but computationally they waste memory better spent solving the problem. As a real world example, consider a PDA P with 7000 labels and 88000 locations. When combined with an NFA containing a single epsilon transition, the resulting augmented PDA P' will contain at least $7000 * 88000 = 616 * 10^6$ transitions. Even at a size of 4 bytes per transition, such a transition system would take up $> 2.4\,\text{GiB}$. In practice, transitions take up much more than 4 bytes.

The PDAs examined in this paper all contain a large number of transitions that either push or swap, and relatively few that pop. A property which makes it possible to closely estimate the set of symbols that can be at the top of the stack, by only looking at the transitions directly connected to the location.

Let the set of symbols that can be at the top of the stack in a state q be denoted $\bar{T}[q]$. The intuition is that a symbol ℓ' can only be at the top of the stack at a location $\ell' \in \bar{T}[q']$ if some transition $(q', \alpha) \in \lambda(q, \ell)$ lead it there where $\alpha = \ell'\alpha'$ if $|\alpha| \geq 1$ or if it was already under the top of stack if $|\alpha| = 0$. Calculating what is under the top of stack can be an expensive operation, so

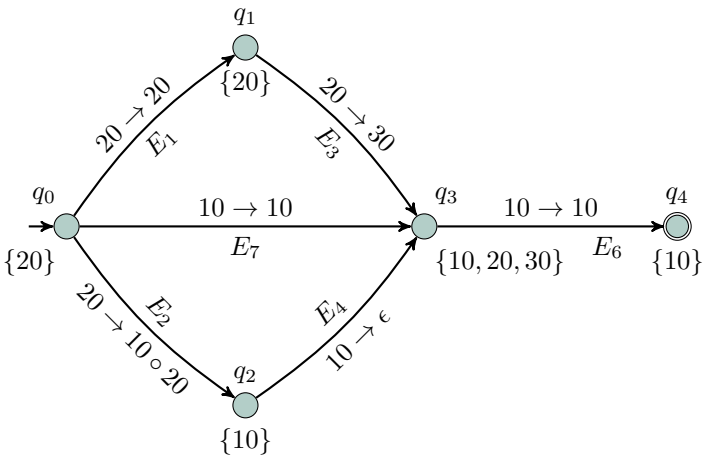


Figure 10: PDA with top of stack annotated

T	q_0	q_1	q_2	q_3	q_4
T_0	$\{20\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
T_1	$\{20\}$	$\{20\}$	$\{10\}$	$\emptyset$	$\emptyset$
T_2	$\{20\}$	$\{20\}$	$\{10\}$	$\{10, 20, 30\}$	$\emptyset$
T_3	$\{20\}$	$\{20\}$	$\{10\}$	$\{10, 20, 30\}$	$\{10\}$
T_4	$\{20\}$	$\{20\}$	$\{10\}$	$\{10, 20, 30\}$	$\{10\}$

Table 6: Value of T_n after n iterations.

we can naively overestimate it to be the entire alphabet Γ. We will call our over-estimation T. An example of a pushdown annotated with T below each location can be found in Fig. 10.

To find the possible top of stack symbols in every location is a slightly tricky task, since the set of non-superfluous incoming transitions for every location depends on the top of stack of the previous locations. The dependency graph will most likely contain cycles, meaning there is no single serializable way to process the nodes once and get a correct answer.

Instead we observe that marking a rule as non-superfluous will only ever result in other rules also being non-superfluous. In other words, adding a label to $T[q]$ for some q, can not cause $T[q']$ to shrink. The problem is therefore to compute the minimum fixed point of the inner for loop which maps current estimate of $T[q]$ for every location, to a new estimate of $T[q]$ for every location.

The function *find_tops* in Algorithm 1 finds the smallest fixed point of function the inner for loop. The associative array returned is our final T for all q.

4.1.1 Usage

Definition 14 (Trimmed pushdown)**.** By examining the $T[q]$ of locations in a pushdown in relation to an ℓ_t, we can eliminate some of the superfluous transitions. We will call the operation *trimming for* ℓ, and denote it $\mathcal{T}(P, \ell)$. Formally, the result of trimming a pushdown $P = (Q, \Gamma, \lambda, q_0, q_f)$ for ℓ_t

```
1  Function find_tops(P, ℓ)
       Input : A PDA  P = (Q, Γ, λ, q_0, q_f),
               An initial label ℓ
       Result: The minimum fixed point T.
2      E_all ← {(q, ℓ, q', α) | (q', α) ∈ λ(q, ℓ)};
3      for q ∈ Q do
4          T_0[q] ← ∅;
5      end
6      T_0[q_0] ← {ℓ};
7      n ← 0;
8      repeat
9          n ← n + 1;
10         T_n ← T_{n−1};
11         for (q, ℓ, q', α) ∈ E_all do
12             if ℓ ∈ T_n[q] then
13                 if |α| ≥ 1 then
14                     T_n[q'] ← T_n[q'] ∪ {head(α)};
15                 else
16                     T_n[q'] ← T_n[q'] ∪ Γ;
17                 end
18             end
19         end
20     until T_n = T_{n−1};
21     return T_n;
22 end
```

Algorithm 1: find_tops(P, ℓ) find the minimal fixed point of T for a given initial label l.

is $\mathcal{T}(P, \ell_t) = (Q, \Gamma, \lambda', q_0, q_f)$. Notice that the conversion only affects the transition function λ. The new trimmed transition function λ' contains all rules $(q', \alpha) \in \lambda'(q, \ell)$ where $(q', \alpha) \in \lambda(q, \ell)$ and $\ell \in T[q]$ where $T = find\_tops(P, \ell_t)$.

Theorem 8. Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and an initial label ℓ_t we construct the trimmed PDA $\mathcal{T}(P, \ell_t) = (Q, \Gamma, \lambda', q_0, q_f)$, satisfying there exists a trace $(q_0, \ell_t w_0) \to_\lambda (q_1, w_1) \to_\lambda \cdots \to_\lambda (q_n, w_n)$ in P if and only if the same trace $(q_0, \ell_t w_0) \to_{\lambda'} (q_1, w_1) \to_{\lambda'} \cdots \to_{\lambda'} (q_n, w_n)$ exists in $\mathcal{T}(P, \ell_t)$.

To prove Theorem 8, we will prove the claim of Lemma 2. Theorem 8 is then trivially true as well.

Lemma 2. Given a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ and an initial label ℓ_t the algorithm $find\_tops$, at step n, has a T such that:

$$\forall n : (q_0, \ell_t w) \to^n_{\lambda'} (q, \ell w') \implies \ell \in T_n[q].$$

Proof. We will prove the above lemma by induction in n.
Base case ($n = 0$): Line 6 in Algorithm 1 adds ℓ_t to T_0.
Inductive case: Assume that Lemma 2 holds for n, we will show that it must hold for $n + 1$. The existence of a trace $(q_0, \ell_t w) \to^{n+1}_{\lambda'} (q', \ell' w')$ implies that $(q_0, \ell_t w) \to^n_{\lambda'} (q, \ell w') \to_{\lambda'} (q', \alpha w)$. Therefore by the induction hypothesis $T_n[q]$ must include the source symbol ℓ, meaning that Line 12 will be true. From here there are two cases, which Line 13 detects. Either α is a string of symbol from which the head is the new top of stack. In that case, Line 14 adds the $head(\alpha)$ to $T_{n+1}[q']$. Otherwise, $\alpha = \epsilon$ and the new top of stack will be what was below ℓ. Since we do not know what was below ℓ Line 16 adds everything to $T_{n+1}[q']$, guaranteeing that anything that could be below ℓ is there. The loop on Line 11 makes sure this is done for all transitions.

□

4.1.2 Example

In Table 6 we can see the progression of T after n iterations of the expansion function, here called T_n, running on the pushdown from Fig. 10. Before the first iteration of the inner for loop Line 11, we have to add the initial label to $T_0[q_0]$. During the first iteration, the inner for loop identifies that E_1 and E_2 are reachable, and adds the head of their respective stack modifications 20 and 10 to the respective destination locations, $T_1[q_1]$ and $T_1[q_2]$, note here that E_7 is not reachable since $10 \notin T_0[q_0]$ as pr. Line 12. The next iteration we discover that E_3 and E_4 are now reachable. For E_3 we again add the head symbol, 30, to destination $T_2[q_3]$, but E_4 is different. Since it pops a label, the stack modification is of length 0, and therefore hits Line 16 which adds the entire alphabet to $T_2[q_3]$. Through another iteration 10 is added to $T_3[q_4]$. T is now at a fixed point, and the iteration can end. For $find\_tops$ to realize that it is a fixed point, it runs another iteration in which nothing is done. Since $T_3 = T_4$, $find\_tops$ knows that it has reached a fixed point, and returns T_4.

4.2 Wildcard Optimization

Recall that a PDA P constructed by Definition 13 never reads from the stack. In our model of a pushdown from Definition 10 the transition function λ must

always read one symbol. To emulate the behaviour of not reading the stack, P contains $|\Gamma|$ transitions for every transitions in N.

To combat explosion of the PDA we will introduce the Optimized Constructing PDA. The key intuition is that our pushdown model makes it possible to rewrite the previous top label while pushing, we call this operation *Push-Replace* and it's represented by a rule like $(q', \ell'\ell'') \in \lambda(q, \ell)$ where $\ell'' \neq \ell$. If we add a unique wildcard symbol at the top of the stack, we can read that in λ, and use the Push-Replace operation to replace the old wildcard a symbol while pushing a new wildcard.

Definition 15 (Optimized Constructing PDA)**.** Given an NFA $N = (S, \Sigma, \delta, s_0, s_f)$, the corresponding optimized constructing PDA $P_o = (Q, \Gamma, \lambda, q_0, q_f)$ is defined as follows:

- $Q = S \cup \{q_0, q_f\}$, where q_0 and q_f are unique start and end locations such that $q_0, q_f \notin S$.
- $\Gamma = \Sigma \cup \{\bot, *\}$ where $*$ is the unique wildcard symbol such that $* \notin \Sigma$,
- λ is the transition function defined below,
- $q_0 = q_0$,
- $q_f = q_f$,

The transition function is defined as follows:

a) P_o will have the rule:

$$(s_0, *\bot) \in \lambda(q_0, \bot),$$

b) P_o will have the rule:

$$(q_f, \epsilon) \in \lambda(s_f, *),$$

c) P_o will have the rule:

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

for every $q, q' \in Q$ and $\ell \in \Sigma$ such that $q' \in \delta^R(q, \ell)$.

d) P_o will have the rule:

$$(q', *) \in \lambda(q, *),$$

for every $q, q' \in Q$ such that $q' \in \delta^R(q, \epsilon)$.

Rule a) pushes the wildcard label $*$ to the top of the stack, where it will remain for until the final location is reached, where rule b) will remove it. During the computation, the pushdown closely matches the behaviour of N, rule c) will replace the old wildcard with the new label ℓ, while pushing a new wildcard to the top of the stack. Rule d) will just emulate the pushdown epsilon rule, maintaining the stack.

Theorem 9. Given an NFA $N = (S, \Sigma, \delta, s_0, s_f)$, we can construct a PDA $P = (Q, \Gamma, \lambda, q_0, q_f)$ such that $(s_0, h_0) \to_\delta^* (s_f, \epsilon)$ if and only if

$$(q_0, \bot) \to_\lambda^* (q_f, h_0\bot)$$

The proof of Theorem 9 is similar to that of Theorem 5, with the exception that rule a) pushes the wildcard, which stays on the top of the stack for the whole computation, and rule b) pops the wildcard as the final step, revealing the built stack w.

5 Constructing the Final PDA

This section will explain how we combine all the previously developed pieces into a single PDA to solve Problem 2.

For an overview of how the combination works see Fig. 11.

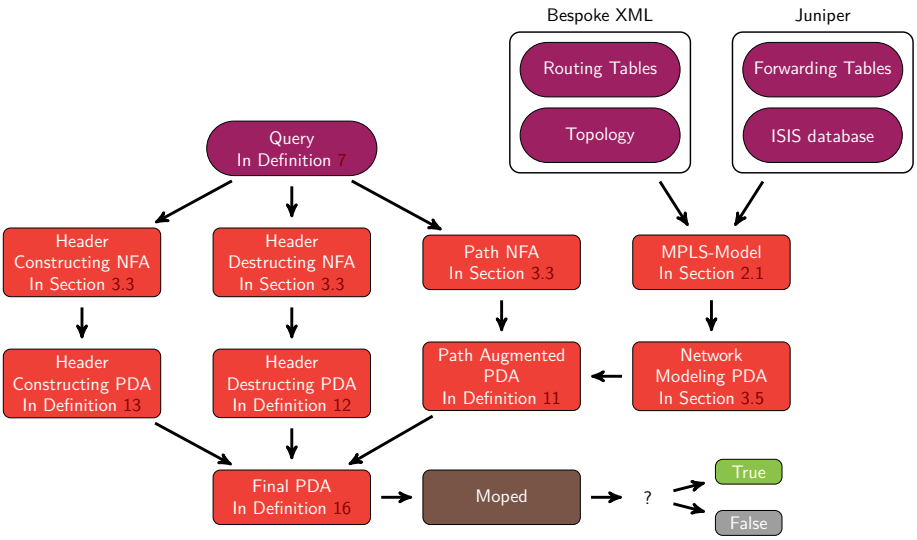


Figure 11: Flowchart of the construction leading to the final PDA.

The construction of this final PDA takes 3 input ingredients:

1. Topology-,
2. routing tables, and
3. a query.

The topology T and routing tables R together make up an instance of the MPLS-model from Section 2.1, which is converted into a PDA encoding the full reachability by following the rules in Section 3.5.

Likewise, the query language from Section 2.4 consisting of four parts, three regular expressions and the maximum number of link failures: 1. header construction, 2. path query, 3. header destruction, and 4. the maximum number of link failures. The first three of these are converted into separate NFAs, per the rules in Section 3.3. The header NFAs, Items 1 and 3 are converted into corresponding PDAs, using the method shown in Section 3.4, and the path NFA is combined with the network modeling PDA using our augmented pushdown construction as shown in Section 3.2.

Finally, we concatenate these three PDAs leaving us with the final PDA, first running through the header construction PDA to make a valid header, then through the path PDA to make sure this header can make it through the network with a valid path, and finally through the destruction PDA to ensure the resulting header is also acceptable.

Definition 16 (Final PDA)**.** Given the header constructing and destructing PDAs $P^{con} = (Q^{con}, \Gamma^{con}, \lambda^{con}, q_0^{con}, q_f^{con})$ and $P^{des} = (Q^{des}, \Gamma^{des}, \lambda^{des}, q_0^{des}, q_f^{des})$, and the path augmented network PDA $P^{aug} = (Q^{aug}, \Gamma^{aug}, \lambda^{aug}, q_0^{aug}, q_f^{aug})$, the final PDA $P^f = (Q^f, \Gamma^f, \lambda^f, q_0^f, q_f^f)$ can be constructed as follows:

- $Q^f = Q^{con} \uplus Q^{des} \uplus Q^{aug}$,
- $\Gamma^f = \Gamma^{con} \cup \Gamma^{des} \cup \Gamma^{aug}$
- λ^f is defined below,
- $q_0^f = q_0^{con}$, and
- $q_f^f = q_f^{des}$.

The transition function λ^f is defined as follows:

a) P' contains the rule

$$(q', \alpha) \in \lambda^f(q, \ell),$$

for all $(q', \alpha) \in \lambda^{con}(q, \ell) \cup \lambda^{des}(q, \ell) \cup \lambda^{aug}(q, \ell)$,

b) P' contains the rule

$$(q_0^{aug}, \ell) \in \lambda^f(q_f^{con}, \ell),$$

c) P' contains the rule

$$(q_0^{des}, \ell) \in \lambda^f(q_f^{aug}, \ell).$$

Rule a) adds all rules from P^{con}, P^{des}, and P^{aug}. Rule b) adds the rule connecting the end of P^{con} to the start of P^{aug}. Rule c) connects the end of P^{aug} to the start of P^{des}.

Theorem 10. Given a topology T, routing tables R, and a query $q = \text{'}<\text{'}\, a \,\text{'}>\text{'}\, b \,\text{'}<\text{'}\, c \,\text{'}>\text{'}\, k$, the constructed PDA $P^f = (Q^f, \Gamma^f, \lambda^f, q_0^f, q_f^f)$ has the property that if there exists some trace in the network $(in_0, h_0), (in_1, h_1), \ldots, (in_n, h_n)$ such that $h_0 \in L(a)$, $h_n \in L(c)$, and $in_0 in_1 \ldots in_n \in L(b)$ for a $|F| \leq k$ there is a trace in P:

$$(q_0^f, \bot) \rightarrow_\lambda^* (q_f^f, \bot).$$

6 Tool Comparison

In order to show that our approach works in practice as well as in theory, we compare our implementation to that of another recognized theory and an implementation thereof. To compare against we have chosen HSA, the reasoning for which is described in Section 6.1 We run these tests on a synthetic network which we can scale, described in Section 6.2, to ensure full control of the network. Additionally we also test Prex on NORDUnet's network, showing that our theory and implementation is viable in real world scenarios. We would have liked to perform similar tests with HSA, however this tool does not scale to such large industrial networks, which is described further in Section 6.4. The results from our tests show that Prex does indeed scale better than HSA, and is useful for real world cases, where HSA is not applicable.

6.1 Choosing Header Space Analysis

In Section 1.3 we presented a variety of tools and their various attributes. Based on the information presented in Table 3, we decided that NetKAT and HSA were the tools best suited for comparison.

NetKAT is part of the Frenetic language suite, for which the focus seems to be more so on providing a language and tool for creating correct network configurations, more so than verifying existing ones. We contacted professors Dexter Kozen and Nate Foster, authors of the NetKAT papers, who confirmed that an implementation of the verification NetKAT system exists, although not suitable for public consumption.

The other suitable option was HSA, which is available at a public repository[1] referenced in the paper, this is not as mature as NetKAT, and abandoned development years ago, leaving it stale. As a result the choice is between two stale tools, given that NetKAT has since then diverged from verification, and since it is not suited for public consumption, we choose HSA as the tool destined for comparison.

6.2 Synthetic Networks

In order to show how both HSA and Prex scales with various attributes, we construct a synthetic network which we can consistently grow. To grow the size of the network we nest this base network into itself. That is, we insert a copy of the original network in the place of a link in the nested network. In our synthetic network the base network is injected in place of the links (v_2, v_3) and (v_2, v_4). This is illustrated in Fig. 12b.

The names of the routers in the sub-networks are mangled to make sure that they are unique. As we scale up the size we perform this nesting recursively i.e. we also inject the base network into the sub-networks. The resulting scaled network ends up containing routers equal to the following formula:

$$|routers_n| = \sum_{i=0}^{n} 9 * 2^i$$

The ingress rules to the nested network are simply forwarding, as a result the maximum label stack does not increase through the nested networks. The scaling affects only the number of routers, thereby increasing the total number of rules, and the length of paths going through edges (v_2, v_3) and (v_2, v_4). The number of rules scales the number of routers as follows:

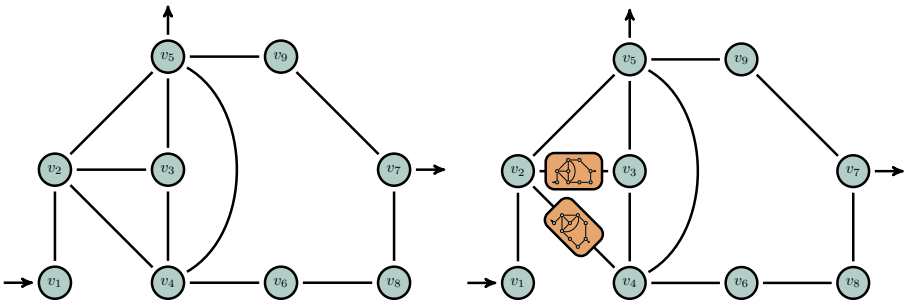
$$|rules_n| = \frac{|routers_n|}{9} \times |rules_0|$$

where $|rules_n|$ is the number of rules at nesting depth n and $|routers_n|$ is the number of routers at nesting depth n. The lengths of paths going through (v_2, v_3) and (v_2, v_4) are increased with at least 5 hops per nesting level, depending on the route taken through the nested network.

6.3 Multiprotocol Label Switching in Header Space Analysis

HSA has two versions, the original python version which was written alongside the paper[29], and an optimised version created in C, the C version being solely for running queries, and not creating the files required to do so. The wiki

[1] https://bitbucket.org/peymank/hassel-public/wiki/Home



(a) Original (b) Nested once

Figure 12: Our base network for building synthetic networks, and one level of nesting.

mentions that the python code is out-dated and only the Cisco IOS parser is needed, this parses the output of 5 Cisco IOS commands to transfer function (TF) files, the files used by HSA. The python code also contain parsers for XML and Juniper, however like the majority of the repository, this does not actually work, only the Cisco parser in relation to the networks from Stanford University, used in the HSA paper for testing and evaluation, works.

6.3.1 Header Space Analysis transfer function

The output from the parsers are in a file format created for HSA, these are known as transfer function files (tf). These files contain all the routing rules for the given router, with 4 fields being the most relevant, *in_ports*, *out_ports*, *match*, and *rewrite*. *in_ports* And *out_ports* designate a specific port on the router, these are equal to what we refer to as interfaces in our model and theory. *match* and *rewrite* contain header representations, *match* for which headers are accepted, equivalent to the accepted label in MPLS, and *rewrite* representing how the header should look when exiting the router, representing the push, pop and swap functions in MPLS.

By design HSA is protocol agnostic, as such headers are simply read as a sequence of bits, it is our job to designate how many bits the header should read, i.e. how much of the header is required in order to do routing. A label in MPLS contains 32 bits of information, as a result the headers we encode in the TF files must be contain $32 * ML$ bits where ML is maximum height of the MPLS label stack.

HSA uses three values to present the headers: x, 0, 1 where x means wildcard, i.e. 0 or 1. An example of a header with $ML = 3$ is shown in Table 7, for this example labels are presented as containing only 8 bits. The example shows how we encode headers as a stack, we reserve a label with the value of only 0 as an empty label field. A full wildcard header presents that any label can occupy the location, this is how we present something that is not top of stack, i.e. we should not discard a header based on anything below the top of stack. Label 2 presents the label containing the binary representation 00001010.

With an understanding of how the headers look, let us have a look at an example of how a rule in the TF format looks, compared to how they look in

Header		
Label 3	Label 2	Label 1
00000000	00001010	$xxxxxxxx$

Table 7: Example of how HSA sees a header with labels containing only 8 bits rather than 32.

Router	In I_v	Label	Prio	Out I_v	Operation
v_3	$v_3^{v_1}$	10	1	$v_3^{v_5}$	swap(11)

Table 8: Snippet of the routing table Table 5 on page 18

our routing table shown in Table 5 on page 18, Table 8 is a snippet of a rule in that table. It the four fields of relevance would be translated as follows:

in_ports The interface $v_3^{v_1}$ is converted to a numerical value as ports must be presented as such. Each router has a unique index in the multiple of 100,000, as such being on router 3 the port may have the value 300,001.

out_ports Similar to in_port $v_3^{v_5}$ is translated to port 300,005.

match The match value would be a constructed header where the accepted label is the only label not constructed of all 0's or all x's. For label 10 this may look like $00000000, 00001010, xxxxxxxx$, again we use an 8 bit label for the sake of readability, the commas separate labels.

rewrite The rewrite value defines how the output header should look, this value can also be empty in case of forwarding rules where the header is unchanged. For the example we have chosen the operation is swap(11) which would create the rewrite header $00000000, 00001011, xxxxxxxx$.

The files created by our parser are missing a few data points, specifically inverse values, which are additional header values. In the paper [29], these are mentioned as being used for infinite loop detection, however an inspection of HSA-C implementation shows that these values are ignored in the implementation. As a result, despite missing those data points our parser produces equivalent output to that of the Cisco IOS parser in terms of HSA functionality.

6.3.2 Header Space Analysis Scaling

In developing a parser to the HSA format, some impactful differences in terms of data were encountered.

With HSA disregarding the meaning of headers, it also disregards the concept of the label stack used in MPLS, as a result, each rule presented in Prex, must be represented in HSA multiple times, to express the possible locations on the stack. In HSA the length of the header, i.e. the maximum number of labels on the stack for MPLS, is decided when creating the data representation. For each rule in Prex R, there must be rules equivalent to the max number of labels ML in the HSA representation. Looking at the example shown in Table 7 we see that the label is onyl matches at a signel position. In order to present that at this location the label 00001010 is accepted, there must exist a rule for each possible location, i.e. as label 3, label 2 and label 1, which equals 3

rules for a header of size 3. The number of rules can therefore be calculated as $R_2 = |R| * ML$, this R_2 refers to that presented in Table 3, which is a scaling factor for reachability queries in HSA.

While unable to perform verification of the full NORDUnet network, the parser can still generate data as TF files, and in doing so, we see the effect of this scaling factor in terms of rules. At the lowest possible header length, i.e. 1 label, the router containing the most in NORDUnet's network has 101,430 rules including fast failover rules. At 1 label the network does not work, if instead we use a more reasonable max header, like 4, the HSA data representation now has 405,720 rules in its config file for this singular router, which as predicted is perfectly linear in scale. If considering SR larger label stacks may be used, e.g. 16 which in turn scales to 1,622,880 rules for a single router, where as it remains 101,430 rules in Prex's representation.

As mentioned the number of rules impacts the reachability runtime which is mentioned in HSA's paper, however, our tests also show that it has a significant impact on the space complexity as well.

6.3.3 Header Space Analysis Under Failure

A core selling point for our approach is that it does not scale exponentially with failures. While not mentioned in the HSA paper [29], this can be expressed. The naive approach is to enumerate all failure combination with up to k link failures, for each k this creates $\binom{n}{k}$ networks, where n is the number of links.

Alternatively, since HSA allows us to encode non-determinism, we can simply add all the fail-over rules to the network, this is the approach we use. In this approach k represents the maximum number of fail over rule a single router can have, i.e. how many links at a single location may fail. This is essentially the same approach we present in Section 3.5.1 on page 26, and in similar fashion it causes the output to be an over approximation. Unlike our approach however, HSA's output contains every single trace that was found, meaning that in post-processing of the data we can remove all the over-approximated traces. This can be done by subsequently removing any trace using the link or links that are supposed to have failed. This allows for some smarter improvements to the naive case, and while this would improve the actual runtime, reducing the over-approximation would still be $\binom{n}{k}$ worst case. As such, we can determine that HSA is indeed exponential in its scaling with queries under failures.

6.4 Header Space Analysis Limits by Implementation

Going from looking at the data and theory presented in the HSA paper [29], to working with the implementation referred to in the same paper, a variety of limits by implementation were discovered.

The implementation has next to no comments, documentation or explanation what so ever, and in multiple areas uses magic numbers, presumably tied closely to the fact that it is designed for Stanford University's network. This is clear throughout the implementation, but strongly in areas where arbitrary limits are imposed by the implementation.

The most significant such limit, is that the HSA-C implementation cannot handle networks larger than 31 routers, due to how the threads are handled in the implementation. Another such limitation is caused by an unexplained

constant named `MAX_APP`, which is set to 10240. Other such implementation specific handling has caused multiple issues while working with HSA, in essence the implementation is developed and optimised so heavily towards Stanford University's network, that without changes it is practically useless in any other industrial, or even synthetic case. These limits are so severe that after weeks of work we still can't get the HSA-C implementation to verify NORDUnet's network.

In order to use the HSA-C implementation as a system capable of testing other networks, we altered the code to change the two aforementioned limitations. The limitation of 31 routers is caused by the implementation using a regular C `unsigned int` in combination with bit-shifting, as a boolean array to handle threads waiting, as a result having more than 31 routers causes a deadlock of threads waiting on a mutex. Changing this value to a `uint64_t` allows for 63 routers, however in order to go beyond that, a major rework of the HSA-C implementation would be required. We sent an email regarding this to the authors of the code, professors Michael Chang and Peyman Kazemian, to confirm that being the case, however they have not responded. As for the `MAX_APP` value, simply increasing it sufficiently stops it from limiting reachability queries. Given that NORDUnet's configuration contains 69 routers, we are still unable to query their full network with HSA, however, these changes does allow for larger synthetic networks.

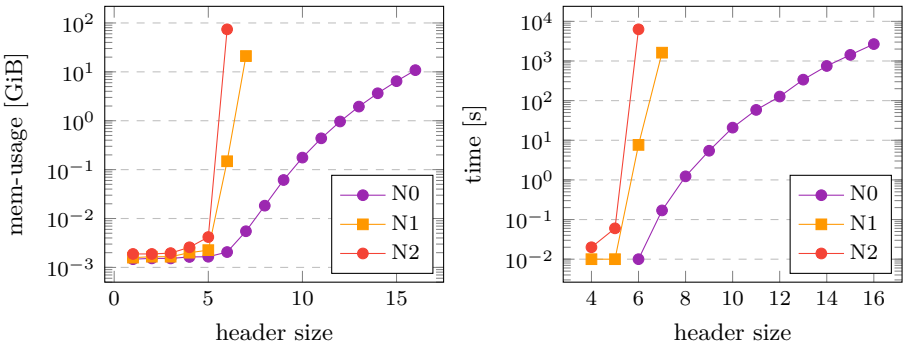
6.5 Header Space Analysis Test Results

In order to test HSA's performance all tests are run on the compute cluster MCC at AAU[2].

All the data is generated by running the same query on the synthetic networks described in Section 6.2. At nesting level 2, the network contains 63 routers, the maximum number of routers that HSA can handle with our modifications. The networks at nesting level 0 and 1 are also used, they contain 9 and 27 routers respectively. Additionally we also scale header size, i.e. the label stack depth, from 1 through 16. The last value we change is k, with each increase in k, a number of fail-over rules are added to the network, we scale this from 0-3, any number higher than 3 would have no effect as it is the maximum number of protection roles encoded in the synthetic network. Combining these factors we reach 192 different networks to run our query on. We run the same query for all the tests, can we from a specific interface on $v1$ reach the a specific interface on $v7$. Each query was allowed 250 GiB, the following plots has certain data points missing this is caused by two things. The query exceeded the memory limit, on the plots these are the values that would go out of frame. The other reason is when values are missing at the beginning, these queries are almost instant, using only a few thousand microseconds.

Figures Fig. 13a and Fig. 13b show 3 graphs presenting the relationship between max header size, time and memory usage. The three graphs represent the different nesting levels used for the test, Nesting 1, Nesting 2, and Nesting 3. Both plots depict similarly scaling graphs showing that the run time and memory usage both scale at the same rate. For all three graphs we see the spike in resources required start around 5, 6 and 7 respectively. For networks with

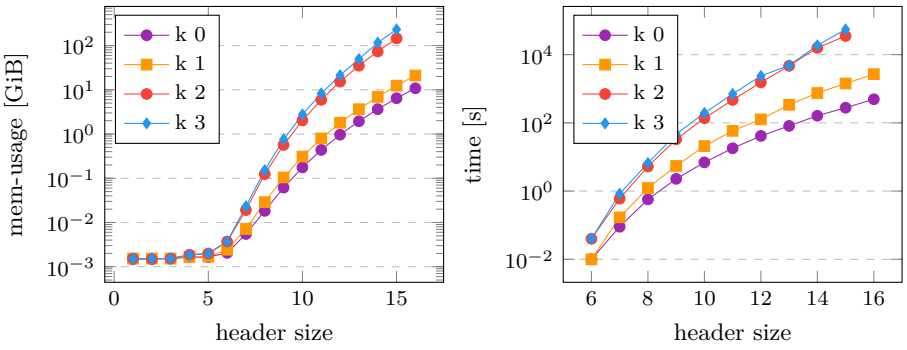
[2] https://sites.google.com/site/mccaau/



(a) memory usage (y-axis) (b) run time (y-axis)

Figure 13: Log scale plot showing the relation between header (x-axis) and another variable for header sizes 1 through 16 with entries for the different nesting factors.

header size 1, 2 and 3 we see very low values, and they are even omitted from Fig. 13b. This is due to no paths existing at such low headers, HSA detects that the input router cannot be exited and terminates, this takes a few thousand microseconds and is thus omitted from the time plot. Scaling the header only adds more rules to the network, where as adding a nesting level adds both rules and routers, thereby forcing longer paths, i.e. more hops, as well as more rules. From the plots we can see that these two increases are not equivalent, thus confirming that the complexity of the network introduced by additional routers, also affects the point to point reachability queries in HSA.

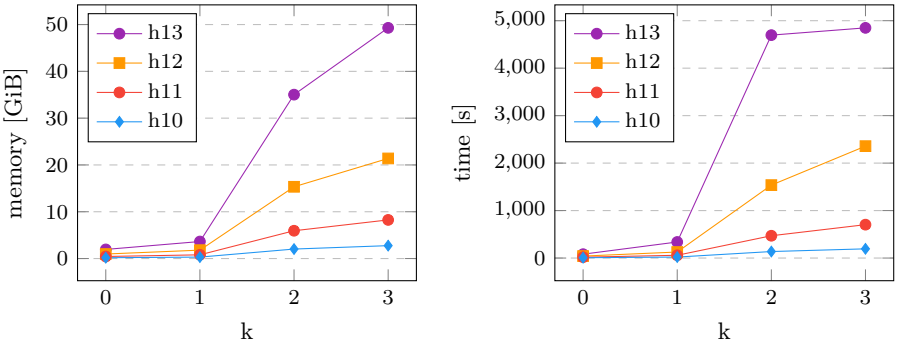


(a) memory usage (y-axis) (b) run time (y-axis)

Figure 14: Log scale plot showing the relation between header and another variable for header sizes 1 through 16 with entries for the different k factors with 0 nesting.

Figures Fig. 14b and Fig. 14a show similar relations, however rather than the graphs presenting the difference in nesting level, they show the impact of changing k. From the plots we can see that even at nesting level zero, that is just nine routers with even < 10 rules per router at max header size of one, the tests run out of resources at max header 12 when adding failure rules for

two or three failures. In contrast, remember that the larger of NORDUnet's routers have 100,000 rules at max header one, and networks realistically need at least a header size of four to work, which would be ~400,000 rules. On top of that, NORDUnet's network contains 69 routers, rather than 9. Given our resource use, and that our largest synthetic network contains only a fragment of the rules NORDUnet's does, yet runs out of resources at maximum header size five, it seems unlikely that HSA would be able to verify any significant queries on NORDUnet's network.



(a) memory usage (y-axis)

(b) run time (y-axis)

Figure 15: Plot showing the relation between k and another variable for header sizes 10 through 13 with nesting 0.

The figures Fig. 15a and Fig. 15b goes to show how unpredictable the impact of changing k is. The impact of changing k will be different from network to network as it adds the number of protection configured in the network. In essence there could exist a protection rule for every single link, which would double the number of rules, or there could be no protection rules in which case changing k has no effect at all. The figures also show that each change to k is irregular, looking at the graphs we see a significantly higher change in going from one to two, than from zero to one or two to three, this further goes to emphasize the unpredictable impact changing k has on resources required for arbitrary networks, given that our three networks contains duplicate rules, changing k has similar but scaled effects.

6.6 Header Space Analysis Conclusion

In conclusion HSA, does not seem to scale exponentially in terms of reachability queries. The data reads as polynomially scaling, albeit a very large polynomial. This is also true for k, but as mentioned this is due to us doing it as on over-approximation when scaling k, the alternative would be to either create an exponential amount of networks, or spend an exponential amount of time on post-processing. When using the over-approximation HSA scales unpredictably on k, as it adds an arbitrary number of rules, based on the network configuration. The most impactful scaling factors are the number of rules, which is what we expected from the paper and working with the data, the length of traces, i.e. rules must be applied to reach the destination, and the number of traces.

HSA / Prex	K: 0	K: 1	K: 2	K: 3
Rules: 23	0.001 / 0.087	0.001 / 0.097	0.003 / 0.093	0.003 / 0.095
Rules: 69	0.005 / 0.253	0.01 / 0.217	0.01 / 0.234	0.06 / 0.218
Rules: 161	0.04 / 0.495	0.06 / 0.534	0.24 / 0.493	0.27 / 0.473
Rules: 345	N/A / 1.067	N/A / 0.991	N/A / 1.009	N/A / 1.037

Table 9: Prex and HSA comparison table for runtime in seconds

HSA / Prex	K: 0	K: 1	K: 2	K: 3
Rules: 23	1.69 / 32.86	1.73 / 32.48	2.03 / 32.70	2.04 / 32.93
Rules: 69	2.31 / 35.63	2.43 / 35.69	4.39 / 35.63	5.62 / 35.68
Rules: 161	4.28 / 41.18	4.79 / 39.79	14.74 / 41.32	16.72 / 40.98
Rules: 345	N/A / 52.86	N/A / 50.42	N/A / 50.58	N/A / 53.06

Table 10: Prex and HSA comparison table for memory usage in MiB

6.7 Prex Benchmark

In this section we present a benchmark our tool Prex. We show how it performs with respect to the query size, the number of routers to be matched by the path query, the maximum number of allowed failures, k, and the size of the network.

6.7.1 Setup

The files containing the network topology and routing configuration are buffered in memory. This isolates the benchmark from the performance of the secondary storage.

As our base network we use the network from Fig. 12a. The queries we run on the network has a randomly generated path query. We generate this path query by randomly selecting the specified amount of routers from the network. For each configuration we run several randomly generated queries select the median, to avoid any outliers to either side.

The queries are as follows:

- The query for scaling the query size

$$< .^* > r_0 .^* r_1 \ldots r_q < .^* > 0$$

  where q is the query size, for all $q \in 50, 100, \ldots, 400$.
- The query for scaling the maximum number of failures

$$< .^* > r_0 .^* r_1 \ldots r_1 0 k$$

where for all k where $0 <= k < 10$

- The query for increasing the nesting level

$$< .^* > r_0 .^* r_1 \ldots r_1 0 < .^* > 0$$

6.7.2 Results

Figure 16a on the following page shows the number of transitions scales linearly in the size of the query, Figs. 16b and 16c shows that the same is true for the total execution time and the peak memory consumption.

Figure 16e is interesting since it shows that number of transitions plateaus as k saturates all protection rules, notice that the change in size is small (y-axis is not starting at 0), therefore we also do not notice this in Figs. 16d and 16f.

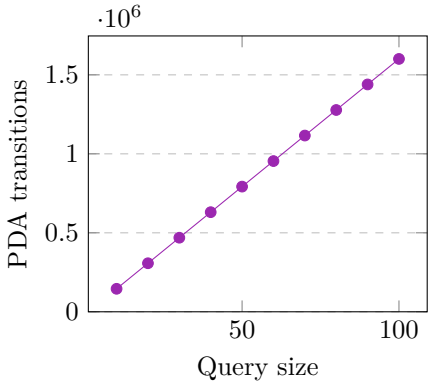
Figure 17a on page 47 shows the linear scaling of transitions wrt. the size of the network, Figs. 17b and 17c validate our belief that the verification is done in polynomial time.

Rules	Query Size	Transitions	Compile (s)	Verify (s)	Total	Peak RSS (KiB)
713	10	145614	18.014	4.527	22.541	349560
713	20	307364	42.611	20.276	62.887	683536
713	30	469025	59.074	21.048	80.122	1021168
713	40	630639	87.236	82.671	169.907	1359628
713	50	792502	103.967	97.551	201.518	1678192
713	60	954166	133	96.723	229.723	2031776
713	70	1115731	138.922	66.017	204.939	2359936
713	80	1277525	168.464	91.178	259.642	2738092
713	90	1439252	196.335	322.44	518.775	3073576
713	100	1600783	210.139	335.298	545.437	3469760

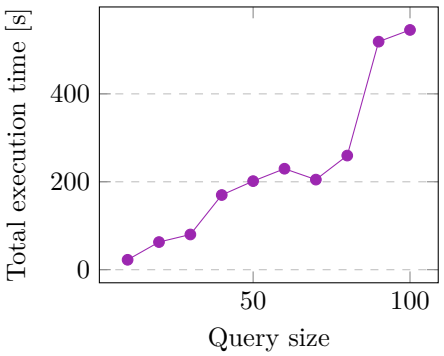
Table 11: Scaling query size

Rules	k	Transitions	Compile (s)	Verify (s)	Total	Peak RSS (KiB)
713	0	145627	18.588	4.397	22.985	348992
713	1	146462	19.409	4.704	24.113	352684
713	2	147651	17.779	3.159	20.938	350584
713	3	148428	18.595	3.165	21.76	345392
713	4	148428	17.765	3.343	21.108	344132
713	5	148450	19.281	3.616	22.897	343280
713	6	148415	17.802	3.246	21.048	343100
713	7	148411	18.894	3.42	22.314	344424
713	8	148444	17.7	3.403	21.103	347116
713	9	148422	19.269	3.44	22.709	345972

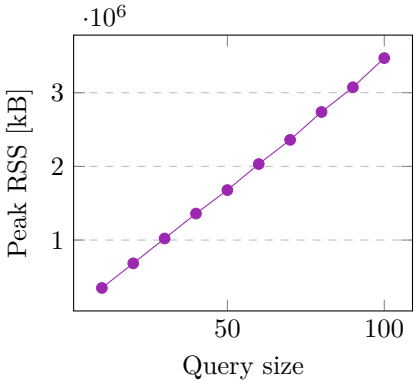
Table 12: Scaling k



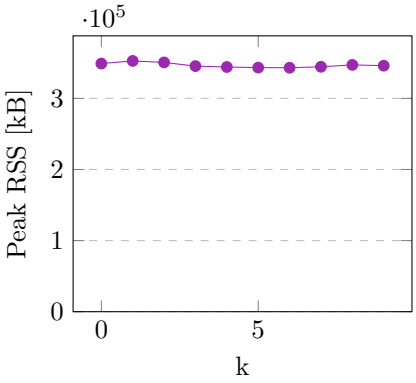
(a) PDS transition count dependence on query size



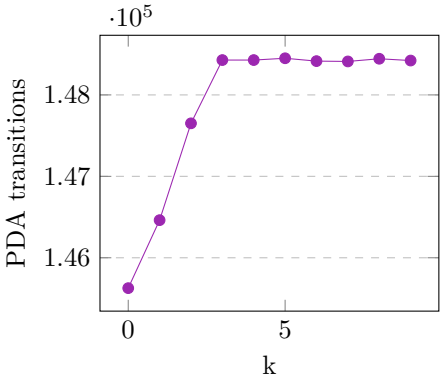
(b) Total execution time dependence on query size



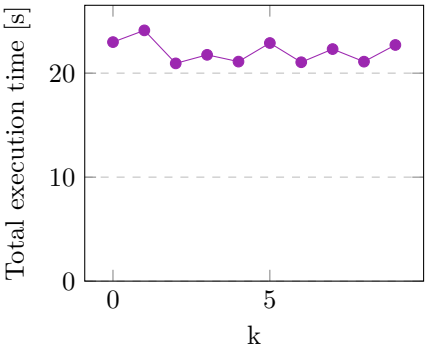
(c) Peak RSS (KiB) dependence on query size



(d) Peak RSS (KiB) dependence on max failures

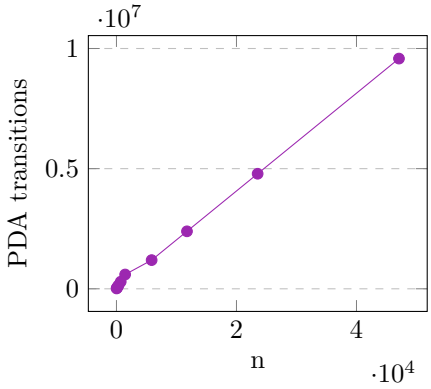


(e) PDS transition count dependence on max failures

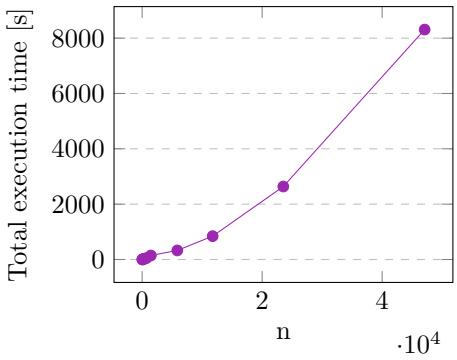


(f) Total execution time dependence on max failures

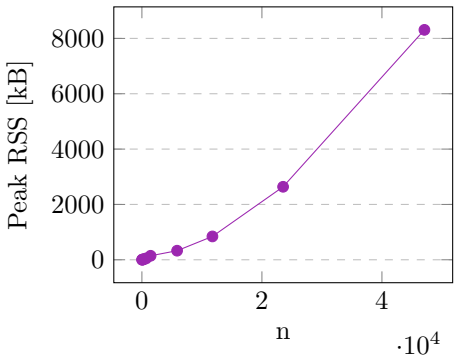
Figure 16: Query size and k scaling test results.



(a) PDS transition count dependence on network size



(b) Total execution time dependence on network size



(c) Peak RSS (KiB) dependence on network size

Figure 17: Nesting scaling test results.

Rules	Nesting Level	Transitions	Compile (s)	Verify (s)	Total	Peak RSS (KiB)
23	1	14623	1.652	0.155	1.807	64116
69	2	33273	3.767	0.451	4.218	108052
161	3	70771	8.812	1.008	9.82	186660
345	4	145579	18.491	4.469	22.96	348044
713	5	295395	35.912	14.505	50.417	657556
1449	6	594905	82.305	62.294	144.599	1321940
5865	7	1193945	152.269	175.392	327.661	2633852
11753	8	2392053	339.391	504.287	843.678	5319116
23529	9	4788193	788.851	1848.959	2637.81	10573728
47081	10	9580486	1814.961	6493.782	8308.743	20883120

Table 13: Scaling Nesting Level

6.8 A Case Study on the NORDUnet MPLS Network

In this section we present a case study we performed on the NORDUnet MPLS network. We present our encoding of their router forwarding tables in our network model. We apply our tool on the network, and show how real questions from NORDUnet can be expressed in our query language, and test how our tool performs under this real-world workload.

All the queries in this section, were run on the AAU MCC cluster[38]. This is a high-performance compute-cluster, with 1TB of RAM per compute node. Running the experiments on the cluster allows us to run many queries in parallel, and have sufficient RAM for very complex queries.

6.8.1 Setup

The NORDUnet MPLS network consists primarily of routers from Juniper, running JunOS. As such we only encode the forwarding tables from their Juniper routers in our network model. The format is documented in depth at [39], here we focus on documenting how we encode the non-obvious parts of the JunOS forwarding tables.

In addition to the forwarding tables, we also use the Intermediate System to Intermediate Systems (IS-IS) database for each router for adjacency information to get information on the network topology. To extract the information from the routers, the following commands needs to be run on each router in the network:

- show route forwarding-table family mpls extensive | display xml
- show isis adjacency detail | display xml

Discrepancies and resolutions between our network model and JunOS

- JunOS only matches on the incoming top label. In our network model we match on the incoming interface as well as the incoming top label. This is resolved by adding the forwarding entries for all interfaces.
- JunOS uses a weight coefficient for rules matching the same label. The meaning of the weight is the priority of the rule, with lower weight being higher priority. The active rule with the lowest weight receives all the traffic. The ordering defined by the weight coefficients is what we use for our sequence of rule sets.
- JunOS supports balancing traffic between multiple rules matching the same label, having the same weight. We do not support balancing and encode these rules as a traffic engineering group.

6.8.2 Network Reachability

As a part of our case study on the NORDUnet MPLS network we performed reachability queries between all pairs of routers in the network. The queries took the form:

$$< .^*. > \text{ from } .^* \text{ to } < .^*. > k, \text{ for } 0 <= k <= 1$$

asking whether we can start in 'from' with any header, go through any number of routers, and leave 'to' with any header. The Tables 14 and 15 structured such that, the row is the starting router and the column is the destination router. Hence the cell 1,2 is the answer to the query:

$$< .^*. > \text{ ch-gva } .^* \text{ de-ffm } < .^*. > k, \text{ for } 0 <= k <= 1$$

The queries were run with the over-approximation, which means that 'YES' is inconclusive and 'NO' means there is no trace in the network satisfying the query.

We found that the network seems strongly connected, as only 'NO' conclusively means that no trace exists. This seems contradictory when we consider all the 'NO' entries in the tables. These 'NO' entries are caused and explained, a lack of forwarding entries in the forwarding tables of the routers, or because the router is connected to the rest of the network through a non-Juniper router, from which we do not have forwarding tables.

Running the queries we saw a large variance in the running time and peak memory use. The total execution time varied between 30 minutes and 90 minutes, and peak memory usage varied between 12GB and 25GB. This is caused by random variations in the intermediary format passed to Moped, and how Moped analyses the pushdown.

6.8.3 Queries on the NORDUnet Network

After discussion with NORDUnet, we came up with some properties they would like tested on their network. We describe these informally and present queries, in our query language see Section 2.4 on page 15, checking for these properties.

1. *Avoid a router or, set of routers, unless they are the destination.*
   NORDUnet were interested in this as they have some locations with lower capacity links. Due to the lower capacity they want to avoid sending traffic over these links unless it has to, i.e. it is destined to the location.
2. *Ensure correct behaviour of service labels.*
   In many instances, NORDUnet use a service label which is used by the last hop router to determine the packets destination outside the NORDUnet network. NORDUnet want to verify that packets with service labels always arrive at their intended destination.
3. *Verify behaviour of node-link protection.*
   JunOS supports node-link protection which protects an LSP bypassing an entire router in the path. This is more extensive than the commonly used link protection, which merely bypasses a link. NORDUnet want to verify this behaviour and investigate how configuring this high-level setting causes the underlying forwarding tables to change.

The above queries can be expressed in our query language as follows:

Table 14: NORDUnet Network Reachability for k=0

	ch-gva	de-ffm	de-hmb	dk-ore	dk-uni	fi-csc2	hk-chw	is-rey	is-rey2	is-vrn	nl-sar	no-nvk	no-usi	no-uts	se-fre	se-lla2	se-tug	se-tug2	uk-hex	us-ash	us-chi	us-man	us-mia	us-pal
ch-gva	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
de-ffm	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
de-hmb	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
dk-ore	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
dk-uni	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
fi-csc2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
hk-chw	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
is-rey	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
is-rey2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
is-vrn	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
nl-sar	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
no-nvk	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
no-usi	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
no-uts	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
se-fre	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
se-lla2	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO
se-tug	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
se-tug2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
uk-hex	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-ash	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-chi	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-man	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-mia	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	YES	NO
us-pal	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

Table 15: NORDUnet Network Reachability for k=1

	ch-gva	de-ffm	de-hmb	dk-ore	dk-uni	fi-csc2	hk-chw	is-rey	is-rey2	is-vrn	nl-sar	no-nvk	no-usi	no-uts	se-fre	se-lla2	se-tug	se-tug2	uk-hex	us-ash	us-chi	us-man	us-mia	us-pal
ch-gva	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
de-ffm	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
de-hmb	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
dk-ore	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
dk-uni	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
fi-csc2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
hk-chw	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
is-rey	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
is-rey2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
is-vrn	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
nl-sar	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
no-nvk	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
no-usi	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
no-uts	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
se-fre	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
se-lla2	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO
se-tug	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
se-tug2	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
uk-hex	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-ash	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-chi	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-man	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	YES	NO	YES	NO	YES	YES	YES	YES	YES	YES	YES	NO
us-mia	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	NO	NO	YES	NO
us-pal	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

1. Avoid router

$$< . * . > [\hat{}routerToAvoid] .^* routerToAvoid .^* [\hat{}routerToAvoid] < . > 1$$

2. Avoid egress

$$< 80001. > ingressRouter\ [\hat{}egressRouter]^* < . > 1$$

3. LSP node-link protect

$$< 200000. > upstream\ [\hat{}protectedRouter]^*\ nextNextHop < . * . > 1$$

The dot present in all 3 queries' constructing and destructing query parts, represent a label in $L^I P$, that is, a non-MPLS label. If this is the only label, the header contains no MPLS labels. The first query can be read as: *Allowing up to 1 failure; Can we enter with any header in any router, that is NOT 'routerToAvoid', and going through any number of routers, reach 'routertoAvoid', then, going through any number of routers, leave any router which is not 'routerToAvoid', with an empty header.* This query can be used to test for the first property. If it answers 'NO' in the over-approximation we cannot leave 'routerToAvoid', and subsequently leave another router with a header containing no MPLS labels.

The second query can be read as: *Allowing up to 1 failure; Can we enter 'ingressRouter' with a header containing '80001' as the only MPLS label, go through any number of routers, which are not 'egressRouter', at any point leaving one with a header containing no MPLS labels.* If this query answers 'NO' in the over-approximation, we know that only the destined 'egressRouter' can pop the service label '80001', In short, the query asks if a router, which is not 'egressRouter', can pop label '80001'. This property require a great deal of knowledge of the network of the LSP allocations in the network. The operator asking the query must determine the service label in question, as well as the two routers it is used between.

The second query can be read as: *Allowing up to 1 failure; Can we enter 'upstream' with a header containing '200000' as the only MPLS label, going through any number of routers, not 'protectedRouter', reach 'nextNextHop' and leave it with any header.* This query must answer 'YES' in the under-approximation to confirm the property. To perform this query, the operator must inspect the LSP, which they protected, to determine the label at the top of the header received by the 'upstream' immediately before the 'protectedRouter' in the LSP.

6.8.4 Running the Queries

Of the 3 properties which NORDUnet expressed interest in, we elected to test for the first property in 3 cases mentioned by them.

- Avoid routing through is-rey and is-rey2
- Avoid routing through no-usi
- Avoud routing through hk-chw

The first query is a bit different because it needs to avoid a set, we shall state it here:

$$< . * . > [\hat{}is\text{-}rey\ is\text{-}rey2] .^* [is\text{-}rey\ is\text{-}rey2] .^* [\hat{}is\text{-}rey\ is\text{-}rey2] < . > 1$$

Table 16: Avoid query results

	Answer	Peak RSS (KiB)	Total Execution Time (s)	Transitions
is-rey	YES	25126592	3433.31	68364566
no-usi	NO	37010336	3418.29	68526634
hk-chw	NO	37086284	3427.89	68463198

Notice that the middle part is a set, instead of a single router.

Table 16 shows the results of the queries. As the queries were run with the overapproximation, the 'YES' answer for 'is-rey' is inconclusive.
What is obvious from Table 14 is that we have an empty forwarding table for 'no-usi', which means the 'NO' result is not applicable. However we include the result as it shows the performance of our tool is affected by the answer and the total network size.
For 'hk-chk' the 'NO' is likely applicable as Table 14 shows that it is possibly reachable.

Through this case study on the NORDUnet MPLS network, we have shown that our query language can be used to express properties, motivated by network engineers, for a real, running network. We have shown that our tool can be applied to this real-world network, and argue through this that Prex can be applied to other real-world networks.

7 Conclusion

Our paper presents the tool Prex, a tool for performing reachability queries on prefix rewriting networks. By expanding the MPLS network model presented by Schmid and Srba [34], we create a more general model capable of expressing features of MPLS networks closely inspired by commercial implementations, e.g. multi-operation support and non-determinism. Based on this model we leverage automata theory to convert a query expressed as a regular expression into a single PDA, which can be used as input to Moped, the pushdown model checker we utilize. The intermediary steps between receiving a query and the final pushdown includes several optimizations which drastically reduces the number of transitions we need to explore. These optimizations help make the tool viable for performing reachability queries on large industrial networks actually used in the industry, as the results from tests on NORDUnet's network show. Our experiments suggest that the runtime of Prex is polynomial in the size of the network.

Furthermore we compared Prex to another recognized theory and tool, HSA. We showed that while the theory for HSA may be sound, the tool that implements it scales very poorly with the label stack, and is simply unable to model larger industrial networks, like that of NORDUnet's. This is in contrast to Prex which is unaffected by the label stack size, and indeed is able to perform queries on NORDUnet's network.

There is still work to be done on Prex, most particularly expanding it with an under-approximation approach, in order to reduce the uncertainty from the over-approximation queries. We believe there is still much work to be done with this framework, but we believe that our work has shown the viability of

this methodology. Additional unexplored approaches to improving the method and tool are also mentioned in Section 7.1.

7.1 Future Work

7.1.1 Wildcard Transitions in Pushdown Solver

For our tool we use Moped, version 1.2 by Schwoon [33], to analyze the reachability in the final PDA. A common occurrence in our NFAs are epsilon transitions, when we construct the augmented PDA, it requires us to enumerate all the labels which can be at the top of stack. We tried to minimize this set of possible labels with our optimizations in Section 4.1, however if Moped supported wildcard transitions, we could avoid having to generate a lot of transitions in our final PDA.

In essence we would like the ability to form rules in the pushdown that acts as follows in a single transition in the PDA:

$$(q', \alpha) \in \lambda'(q, *)$$

where $|\alpha| \leq 2$ and $\alpha = \ell \circ *$, for some $\ell \in \Gamma$, or $\alpha = *$, or $\alpha = \epsilon$.

Transitions added by rule a) on page 21 as part of the Augmented Pushdown Construction are an example of some that could reduced. Since these add $|\Gamma|$ transitions for every ϵ-transition in the NFA.

7.1.2 CEGAR

We would like to implement the CEGAR technique of "Counterexample-guided abstraction refinement for symbolic model checking"[40] to our approach. We would apply it to minimize the alphabet of the pushdown, as the time it takes to verify reachability scales with the number of transitions, and a smaller alphabet will often result in fewer transitions.

7.1.3 Synthesis of Network configuration

Given a network topology and some high level goals, i.e. where to have connectivity, a minimum number of fast-reroute paths, etc. One could use the tool developed in this paper in combination with other approaches to perform synthesis of a routing table. The general way this would work is that the high level goals would be transformed into a set of queries for the tool, then generate some routing table and use our tool check if all the queries are satisfied, otherwise try again.

7.1.4 Further Top of Stack Set Reduction

We think that it is possible to perform further improvements to our Top of Stack reduction in Section 4.1 on page 30. Specifically it might be possible to reduce the size of $T[q]$ when a pop occurs. In Line 16 on page 32 we potentially over-approximate the labels which can be at the top of the stack following a pop.

For example in Fig. 10 and Table 6 we evaluate $T_4[q_3] = \{10, 20, 30\}$ with 20 as the initial header starting at q_0. $T_4[q_3] = \{10, 20, 30\}$ since E_4 pops the stack, and this adds the entire Γ. This is an over-approximation, since 10

cannot be at the top of the stack at q_3, when 20 is the initial header. We imagine that one could keep track of what can be under the top of stack, by keeping another set of labels at each location in the PDA. Then include the computation of this in the *find_tops* and *expand_top* functions, and only terminating when they also reach a fixed point. Also changing Line 16 on page 32 to be $T[q'] \leftarrow T[q'] \cup U[q]$, where U is the set of what can be under the top of stack, such that the information about what could be under the top of stack at the previous location, is included in the possible top of stack labels of q'.

7.2 Acknowledgments

We would like to extend our thanks and gratitude, to the good people of NORDUnet: Magnus Bergroth, Markus Krogh, Henrik Thostrup Jensen, and Dennis Wallberg, for all their assistance with furthering our understanding of MPLS in practice, and providing data for our case study.

We would like to thank Peter Gjøl Jensen and Mads Boye both of AAU, for assisting us with getting access to the AAU MCC cluster.

Lastly we would like to thank our supervisors Jiri Srba and Stefan Schmid.

References

[1] Vince Fuller and Tony Li. *Classless Inter-Domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan*. URL: https://tools.ietf.org/html/rfc4632 (visited on Mar. 23, 2018).

[2] Arun Viswanathan et al. *Multiprotocol Label Switching Architecture*. URL: https://tools.ietf.org/html/rfc3031 (visited on Feb. 19, 2018).

[3] Lou Berger et al. *RSVP-TE: Extensions to RSVP for LSP Tunnels*. URL: https://tools.ietf.org/html/rfc3209 (visited on Mar. 19, 2018).

[4] George Swallow et al. *Fast Reroute Extensions to RSVP-TE for LSP Tunnels*. URL: https://tools.ietf.org/html/rfc4090 (visited on Mar. 19, 2018).

[5] Juniper. *Understanding Source Packet Routing in Networking (SPRING)*. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/source-packet-routing.html.

[6] *RFC Draft: Segment Routing Architecture*. URL: https://tools.ietf.org/html/draft-ietf-spring-segment-routing-15.

[7] *RFC7855: Source Packet Routing in Networking (SPRING) Problem Statement and Requirements*. URL: https://tools.ietf.org/html/rfc7855.

[8] *RFC Draft: IS-IS Extensions for Segment Routing*. URL: https://tools.ietf.org/html/draft-ietf-isis-segment-routing-extensions-15.

[9] *RFC Draft: OSPF Extensions for Segment Routing*. URL: https://tools.ietf.org/html/draft-ietf-ospf-segment-routing-extensions-24.

[10] *RFC Draft: OSPFv3 Extensions for Segment Routing*. URL: https://tools.ietf.org/html/draft-ietf-ospf-ospfv3-segment-routing-extensions-10.

[11] Cisco Systems. *Cisco Segment Routing Configuration Guide*. 2017. URL: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/seg_routing/configuration/xe-16/segrt-xe-16-book.pdf (visited on Mar. 15, 2018).

[12] Stephane Litkowski et al. *RFC draft: Topology Independent Fast Reroute using Segment Routing*. URL: https://tools.ietf.org/html/draft-bashandy-rtgwg-segment-routing-ti-lfa-02 (visited on Mar. 14, 2018).

[13] Abdelali Ala et al. "Core Backbone Convergence Mechanisms and Microloops Analysis". In: 3 (July 2012).

[14] Juniper. *Fast Reroute Overview - Technical Documentation - Support - Juniper Networks*. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/mpls-fast-reroute-overview.html (visited on Mar. 16, 2018).

[15] Stefano Previdi et al. *RFC draft: Segment Routing Architecture*. URL: https://tools.ietf.org/html/draft-ietf-spring-segment-routing-15 (visited on Feb. 19, 2018).

[16] Klaus-Tycho Foerster et al. "TI-MFA: Keep Calm and Reroute Segments Fast". In: *Proc. IEEE Global Internet Symposium (GI)*. 2018.

[17] Anduo Wang et al. "FSR: A Formal Analysis and Development Toolkit for Safe Inter-Domain Routing". In: (Apr. 5, 2018).

[18] Shivkumar Muthukumar et al. "Declarative Toolkit for Rapid Network Protocol Simulation and Experimentation". In: (Apr. 5, 2018).

[19] *The Yices SMT Solver*. URL: http://yices.csl.sri.com/ (visited on Apr. 5, 2018).

[20] Ari Fogel et al. "A General Approach to Network Configuration Analysis". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 469–483. ISBN: 978-1-931971-21-8. URL: http://dl.acm.org/citation.cfm?id=2789770.2789803 (visited on Apr. 5, 2018).

[21] Ryan Beckett et al. "Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations". In: Aug. 22, 2016, pp. 328–341. DOI: 10.1145/2934872.2934909.

[22] *Propane Language*. URL: https://propane-lang.org/ (visited on Apr. 5, 2018).

[23] Ryan Beckett et al. "Network Configuration Synthesis with Abstract Topologies". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 437–451. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062367. URL: http://doi.acm.org/10.1145/3062341.3062367 (visited on Apr. 5, 2018).

[24] Ahmed El-Hassany et al. “NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/el-hassany.

[25] Armando Solar-Lezama et al. “Combinatorial Sketching for Finite Programs”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 404–415. ISBN: 978-1-59593-451-2. DOI: 10.1145/1168857.1168907. URL: http://doi.acm.org/10.1145/1168857.1168907 (visited on Apr. 5, 2018).

[26] Ahmed El-Hassany et al. “Network Wide Configuration Synthesis”. In: *29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Computer Aided Verification. Ed. by Ahmed El-Hassany et al. Vol. 10427. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-63390-9. URL: http://link.springer.com/10.1007/978-3-319-63390-9 (visited on Mar. 21, 2018).

[27] Carolyn Jane Anderson et al. “NetKAT: semantic foundations for networks”. In: ACM Press, 2014. ISBN: 978-1-4503-2544-8. URL: http://dl.acm.org/citation.cfm?doid=2535838.2535862 (visited on Feb. 5, 2018).

[28] Ahmed Khurshid et al. “VeriFlow: Verifyin Network-Wide Invariants in Real Time”. In: p. 13. URL: https://people.csail.mit.edu/alizadeh/courses/6.888/papers/veriflow.pdf (visited on Mar. 5, 2018).

[29] Peyman Kazemian et al. “Header Space Analysis: Static Checking For Networks”. In: p. 14. URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final8.pdf (visited on Mar. 1, 2018).

[30] Haohui Mai et al. “Debugging the data plane with anteater”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. ACM, 2011, pp. 290–301.

[31] Hing Leung. “Regular Languages and Finite Automata”. In: *AMC* 10 (2010), p. 12.

[32] Stephen Cole Kleene. *Representation of events in nerve nets and finite automata*. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[33] Stefan Schwoon. “Model-checking pushdown systems”. In: (2002).

[34] Stefan Schmid and Jiri Srba. “Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks”. In: *Proc. IEE INFOCOM*. 2018.

[35] Juniper. *MPLS Overview*. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/mpls-security-overview.html.

[36] Ken Thompson. “Programming Techniques: Regular expression search algorithm”. In: *Communications of the ACM* 11.6 (June 1, 1968), pp. 419–422. ISSN: 00010782. DOI: 10.1145/363347.363387. URL: http://portal.acm.org/citation.cfm?doid=363347.363387 (visited on Mar. 19, 2018).

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

[38] AAU. *Hardware Organisation - mccaau*. URL: https://sites.google.com/site/mccaau/home/hardware (visited on June 7, 2018).

[39] Juniper. *show route forwarding-table - Technical Documentation - Support - Juniper Networks*. URL: https://www.juniper.net/documentation/en_US/junos/topics/reference/command-summary/show-route-forwarding-table.html (visited on June 7, 2018).

[40] Edmund Clarke et al. “Counterexample-guided Abstraction Refinement for Symbolic Model Checking”. In: *J. ACM* 50.5 (Sept. 2003), pp. 752–794. ISSN: 0004-5411. DOI: 10.1145/876638.876643. URL: http://doi.acm.org/10.1145/876638.876643.