Quantitative Verification and Synthesis of Resilient Networks

Schou, Morten Konggaard

DOI (link to publication from Publisher):
10.54337/aau588616986

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

QUANTITATIVE VERIFICATION AND SYNTHESIS OF RESILIENT NETWORKS

BY
MORTEN KONGGAARD SCHOU

DISSERTATION SUBMITTED 2023

AALBORG UNIVERSITY
DENMARK

Quantitative Verification and Synthesis of Resilient Networks

Ph.D. Dissertation
Morten Konggaard Schou

Dissertation submitted August, 2023

Dissertation submitted: August, 2023

PhD supervisor:: Professor Jiří Srba
Aalborg University

PhD committee: Associate Professor Michele Albano (chair)
Aalborg University, Denmark

Associate Professor Laurent Vanbever
ETH Zürich, Switzerland

Senior Researcher Nikolaj Bjørner
Microsoft Research, USA

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-659-1

Published by:
Aalborg University Press
Kroghstræde 3
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Stibo Complete, 2023

Abstract

Computer networks connect people across the world and are a critical infrastructure for many of the services that a modern society depends on. With the rapidly growing use of computer networks, the demand for higher bandwidth and lower latency to support a new generation of applications, as well as the need for failure protection mechanisms to meet the increasingly stringent dependability requirements, modern computer networks have evolved into complex heterogeneous systems. These networks are tricky to manage and operate correctly, and there are many examples of subtle configuration errors taking down entire networks.

Formal methods have been proposed as a way to reduce the risk of network outages by creating mathematically well-defined models of the networks and their behavior and applying algorithmic techniques to verify conformance with the specifications or synthesize correct-by-construction configurations.

In this thesis, we contribute to the formal treatment of the widely-deployed MPLS networks with a focus on quantitative properties and resiliency mechanisms. We present a formal model of the MPLS network data plane and use this model to develop a practical technique for increasing the resilience of MPLS networks by synthesizing loop-free recursive failover protections. Moreover, we extend prior work on model checking MPLS networks with failover protections to also consider quantitative properties with shortest and longest trace analysis. This work builds on the connection between MPLS networks and pushdown automata. To address the scalability of MPLS network model checking, we significantly improve the performance of pushdown automata reachability checking by developing new on-the-fly algorithms and an efficient tool implementation. We increase the trustworthiness of the model checking results by formally proving the correctness of the used algorithms in the proof assistant Isabelle/HOL, extracting executable code, and performing differential testing against this formally verified oracle.

Finally, we address the importance of providing accurate data for the formal models through a case study with a network analytics company, where we present a robust and efficient method for inferring the otherwise unknown ratios that the routers use when load balancing traffic among multiple paths.

Resumé

Computernetværk forbinder mennesker over hele verden og er en kritisk infrastruktur for mange af de tjenester, som et moderne samfund afhænger af. Med den hastigt voksende brug af computernetværk, efterspørgslen på højere båndbredde og lavere latenstid til at understøtte en ny generation af applikationer, samt behovet for fejlbeskyttelsesmekanismer for at opfylde de stadigt strengere krav til pålidelighed, har moderne computernetværk udviklet sig til komplekse heterogene systemer. Disse netværk er vanskelige at administrere og operere korrekt, og der er mange eksempler, hvor subtile konfigurationsfejl har fået hele netværk til at gå ned.

Formelle metoder er blevet foreslået som en måde at reducere risikoen for netværksafbrydelser ved at skabe matematisk veldefinerede modeller af netværkene og deres adfærd, og anvende algoritmiske teknikker til at verificere overensstemmelse med specifikationerne eller syntetisere konfigurationer, der er korrekte per konstruktion.

I denne afhandling bidrager vi til den formelle behandling af de vidt udbredte MPLS-netværk med fokus på kvantitative egenskaber og resistensmekanismer. Vi præsenterer en formel model af MPLS-netværkets dataplan og bruger denne model til at udvikle en praktisk teknik til at øge modstandsdygtigheden af MPLS-netværk ved at syntetisere fejlbeskyttelse med løkkefri rekursive reserve-veje. Desuden bygger vi videre på tidligere arbejde med modelkontrol af MPLS-netværk med fejlbeskyttelse, og udvider til også at verificere kvantitative egenskaber med analyse af korteste og længste vej. Dette arbejde bygger på forbindelsen mellem MPLS-netværk og pushdown-automater. For at håndtere skalerbarheden af MPLS-netværksmodelkontrol, reducerer vi markant køretiden af reachability analyse i pushdown-automater ved at udvikle nye on-the-fly-algoritmer og en effektiv implementering. Vi øger troværdigheden af resultaterne fra modelkontrollen ved formelt at bevise korrektheden af de anvendte algoritmer i bevis-værktøjet Isabelle/HOL, udtrække en eksekverbar kode og udføre differentiel testning mod dette formelt verificerede orakel.

Endeligt arbejder vi med vigtigheden af at levere præcise data til de formelle modeller gennem et casestudie med et netværksanalysefirma, hvor vi præsenterer en robust og effektiv metode til at udlede de ellers ukendte værdier, som routerne bruger, når de belastningsfordeler data trafik mellem flere veje.

Contents

Abstract iii

Resumé v

Preface xi

I Introduction 1

1 Network Verification and Synthesis 4
2 Challenges in Application of Formal Techniques 5
3 Overview of Computer Networks 7
4 Related Work 8
4.1 Verification of Traditional Networks 9
4.2 Verification of Programmable Networks 10
4.3 Traffic Engineering Optimization 11
4.4 Synthesis of Network Configurations and Updates 12
5 Contributions 13
6 MPLS Network Resilience 14
6.1 MPLS Network Model 14
6.2 Data Plane Generation and Simulation 18
6.3 Recursive Fast Reroute Protection 19
7 Pushdown Automata Reachability 21
7.1 Pushdown Automata Reachability for MPLS Verification 21
7.2 Solving Pushdown Reachability Efficiently 25
7.3 Weighted Pushdown Automata Verification 29
7.4 Formal Correctness of Pushdown Verification 33
8 Measuring Real Networks 35
9 Conclusion 38
References 39

II Papers **51**

A R-MPLS: Recursive Protection for Highly Dependable MPLS Networks **53**
1 Introduction . . . 55
2 MPLS Network Model . . . 57
3 R-MPLS Protection . . . 60
3.1 Protectable Forwarding Entries . . . 61
3.2 Loop Avoidance . . . 62
3.3 R-MPLS Algorithm . . . 63
3.4 Recursive Link and Node Protection . . . 65
3.5 Distributed R-MPLS Implementation . . . 66
3.6 Properties of the R-MPLS Protection . . . 67
4 Evaluation of R-MPLS . . . 68
4.1 MPLS Generation and Simulation . . . 68
4.2 Methodology . . . 69
4.3 Results of RSVP Experiments . . . 71
4.4 Results of LDP Experiments . . . 75
5 Discussion . . . 77
6 Related Work . . . 77
7 Conclusion . . . 79
References . . . 80
A Proofs for Section 3.6 . . . 84
B Elaboration on Section 4.1 . . . 86
C Artifact Appendix . . . 88

B MPLS-Kit: An MPLS Data Plane Toolkit **93**
1 Introduction . . . 95
2 MPLS Network Operation . . . 97
3 MPLS-Kit Overview . . . 98
4 MPLS Dataplane Generation . . . 100
5 MPLS Forwarding Simulation . . . 103
6 Use cases . . . 105
7 Conclusions . . . 107
References . . . 107

C Faster Pushdown Reachability Analysis with Applications in Network Verification **111**
1 Introduction . . . 113
2 Preliminaries . . . 115
3 Formal Model of MPLS Networks . . . 116
3.1 MPLS Network Verification . . . 119
3.2 From Query Satisfiability to Pushdown Reachability . . . 120

4 Improving Pushdown System Reachability Analysis 122
4.1 Early Termination of Reachability Algorithms 122
4.2 Combining Forward and Backward Search 123
4.3 Abstraction Refinement for Pushdown System Reachability 124
5 Implementation and Experiments 126
6 Conclusion 128
References 129
A Appendix 131

D AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks 135
1 Introduction 137
2 MPLS Network Model 139
2.1 Network definition 139
2.2 Valid MPLS headers 140
2.3 Example network 141
2.4 Network traces 142
2.5 Query language 143
3 Quantitative Extension 144
4 Tool Implementation 146
4.1 Verification methodology 147
4.2 Tool architecture 147
5 Performance Evaluation 149
6 Conclusion 151
References 151
A Appendix 154

E PDAAAL: A Library for Reachability Analysis of Weighted Pushdown Systems 157
1 Introduction 159
2 Weighted Pushdown Systems and Reachability 160
3 Implemented Algorithms and PDAAAL Architecture 161
4 Comparison with State-of-the-Art 162
5 Applications 163
6 Conclusion 164
References 164

F Differential Testing of Pushdown Reachability with a Formally Verified Oracle 167
1 Introduction 169
2 Isabelle/HOL 172
3 Pushdown Reachability 174
3.1 Nondeterministic pre^* Saturation 174

3.2 Nondeterministic $post^*$ Saturation . . . 177
3.3 Combined $dual^*$ Saturation . . . 180
4 Executable Pushdown Reachability . . . 180
5 Differential Testing . . . 182
5.1 Differential Testing of Pushdown Reachability . . . 183
5.2 Automatic Counter-Example Minimization . . . 184
6 Case Study: Analysis of PDAAAL . . . 185
6.1 Methodology of Test Case Generation . . . 186
6.2 Results . . . 186
7 Conclusion . . . 188
References . . . 188

G Discovery of Flow Splitting Ratios in ISP Networks with Measurement Noise 193
1 Introduction . . . 195
2 Problem Formalization . . . 197
2.1 Network, Paths and Flows . . . 198
2.2 Correlation of Traffic Flow and Link Utilization . . . 199
3 Solution to Flow Splitting Ratio Synthesis . . . 200
3.1 Encoding of FSR to a Linear Program . . . 200
3.2 Measurement Noise . . . 204
4 Dealing with Measurement Noise . . . 204
4.1 Simulation Experiments with Synthetic Traffic . . . 205
5 Scalability Study on Large European ISP . . . 206
6 Conclusion . . . 207
References . . . 208

Preface

While being a PhD student can sometimes feel like a lonely endeavor, the truth is that a lot of brilliant people have helped me throughout this journey, and I believe that a few words of acknowledgement are in order.

First and foremost, I would like to thank my supervisor Jiří Srba for his guidance and extensive feedback throughout this PhD project.

During my five month in Berlin, thanks to Ingmar Poese, I gained valuable insight into the practical challenges of working with data from real networks. I would like to thank all the colleagues at Benocs for quickly making me feel part of the company and for also joining social activities outside of work.

Stefan Schmid has been involved in my PhD project from the beginning, and he has throughout provided me with much-needed guidance on networking research, for which I am grateful. During my stay in Berlin he kindly invited me to his research group at TU Berlin.

With Juan Vanerio, I have had many fruitful discussions, where the combination of our different technical backgrounds has provided some interesting and valuable outcomes. Peter Gjøl Jensen has been a great inspiration, helped guide my intuition for implementing high-performing algorithms, and showed me the value of high-quality code. Anders and Dmitriy, thank you for the collaboration on formalizing pushdown automata and for making me further appreciate the art of interactive theorem proving.

I also want to thank my office mates and the lunch group at DEIS for all the various discussions about life, science, or whatever is currently going on in the world.

Finally, a huge thank you to my parents, Helle and Henning, who initially sparked my curiosity for science and technology, have inspired me to explore my ideas and make a fair effort, and have continuously supported me and shown interest in my research.

Morten Konggaard Schou
Aalborg University, August 8, 2023

Part I

Introduction

Computer networks, like enterprise networks and in particular the internet, have become a ubiquitous part of everyday life and a critical component in the infrastructure of our modern, digital society. Digitalization of core parts of the society like the economy, media, hospitals, and education depend on reliable inter-connection of computers. Even social interaction and entertainment increasingly flow through these networks of routers and switches connected across the world by copper or fiber-optic cables.

The last couple of decades have seen a drastic increase in the use of the internet, and with that an increased demand on the capacity of the networks. Not only do networks need to transport huge amounts of data packets, they need also be able to react to sudden increases or shifts in traffic patterns, as recently witnessed during the COVID-19 lockdowns: people staying at home caused significant shifts in the usage of internet traffic, with an increase in applications for remote work and education like video conferencing and VPN services, as well as entertainment including video-on-demand and online gaming [52].

Most networks, using the internet protocol, have traditionally been optimized for throughput and resource utilization, with less emphasis on decreasing the latency [24]. However, for many of these new types of networked applications, the quality of experience depends heavily on the delays that the user experiences, which in turn brings strict latency requirements on the networks. Loss of connectivity, even if short-lived, can disrupt the user applications. Next generation of networked applications like factory automation, robotics, intelligent transport systems, and tele-surgery will place even further demands for reliability and low latency on modern communication networks [127].

The introduction of these new latency-sensitive applications along with the need for more bandwidth to serve the increasing amount of traffic pushes network operators to evolve their networks, gain more fine-grained control, and steer the traffic flows through the network. With this incremental growth, computer networks evolve into large heterogeneous systems, where a variety of different hardware and software run several protocols with complex configurations—all making the networks difficult to manage and operate correctly.

There are many examples of network outages, where an erroneous network configuration update [78, 107, 150, 159] or a latent software or configuration error being triggered inadvertently under normal operation [8, 132, 139] have taken down large networks and impacted millions of users as well as critical services like emergency phone lines [107, 132] and payment systems. These incidents are often caused by human errors [13, 25, 150] and can even be as simple as a typo in a command [7]. In some cases, the errors also take down internal monitoring tools, which extends the outage by making it harder for the network administrators to locate and fix the issue [8, 78, 107, 132, 159].

To make matters worse, networking equipment is not perfect and will even-

tually fail, so given the magnitude of modern networks, equipment failures are not uncommon. Networks need to be build and configured in a way, so they are resilient to these failures: the failure of an individual component should only have a minimal effect on the continued functioning of the network, including the quality of service aspects like the latency and bandwidth. Further adding to the complexity is an increase in failure scenarios that affect multiple network elements simultaneously [134].

We contribute to the recent efforts to cleverly automate parts of the network operation and use formal models and algorithms to verify its correctness in order to reduce the risk of the network outages that regularly impact millions of people worldwide.

1 Network Verification and Synthesis

One of the basic tools for the management of computer networks is network monitoring [105], which gives network operators the insight into the current behavior of the network and allows them to identify issues and reactively reconfigure the network. A purely reactive and ad-hoc approach to network operation has the risk of introducing errors that quickly lead to network outage before the operator is able to react. Instead, operators can use simulation or emulation to test the configuration changes before deployment [110], inspect the results, and fix any issues observed in the sandbox environment.

While simulation and testing can catch errors early, it cannot check all scenarios that the network might experience, so even if the simulation and testing is done thoroughly, this approach leaves room for implementation and configuration errors in uncommon scenarios, which may lead to severe failures during the lifetime of the system.

Formal verification takes a different, complementary approach to system correctness by constructing a formal, mathematically well-defined model of the system and its specification. Model checking [36, 37] can then be used to exhaustively verify that the system satisfies the specification in all possible situations.

Model checking comes with the problem of state-space explosion, since most systems have at least an exponential, if not unbounded, number of possible inputs to check. Much work has been done in the field of formal methods to tackle this state-space explosion for various different modelling formalisms by constructing efficient model checking algorithms that still guarantee a full correctness of the results. This correctness guarantee, however, only holds if the model checker itself is correct. Confidence in model checking results, e.g. when used in safety-critical systems, can be increased by certification, extensive testing, algorithm review or combination with theorem proving efforts [102, 121].

Formal synthesis takes it a step further. Instead of verifying the correctness of a given system model, synthesis takes a specification and generates a system that is correct by construction. On top of the Boolean correctness according to a specification, synthesis may further try to optimize quantitative objectives—creating a system that is not only correct but also efficient.

When it comes to computer networks, the specification of desired properties can include qualitative correctness and safety properties as well as quantitative performance properties. Correctness properties are e.g. that packets can reach their destination; that all traffic flows through a certain waypoint, such as a firewall; the absence of forwarding loops; or the absence of "black holes", where packets are dropped because the router has no forwarding rule for it.

Quantitative properties are relevant for the performance of computer networks and include for instance the maximum latency experienced by packets on a given traffic flow, the utilization of each link given typical traffic demands, and the maximum throughput the network can serve across different traffic flows given the bandwidth capacities of each link. As traffic patterns change during the day, many of the quantitative properties change with them, so network performance measures need to also take this temporal variation into account.

With the possibility of failures of individual components and the changes induced by the network's reaction to a failure, these properties need to be seen relative to their resilience. Given estimates of the probability of various failures, we can use the notion of availability to express the probabilistic guarantee that a given property is met at least a certain percentage of the time.

2 Challenges in Application of Formal Techniques

Developing formal techniques for the modelling, verification, and synthesis of computer networks is a significant undertaking. We will here consider six of the major challenges that should be addressed to enable formal verification and synthesis becoming an essential tool for making modern computer networks safe, reliable, and efficient.

Network formalization: Errors can happen at different layers, in the configuration of many different protocols, and in their interplay. An error that manifests in the data plane—the forwarding rules installed at each router, responsible for forwarding packets—may be caused by an incorrect configuration at the control plane. This indicates that a complete formal model can become quite complex in order to incorporate all essential elements of a modern network. Finding the right abstraction and formalization for each part, as well as creating composable models may be the key to solving this challenge.

Network mining: Precise formal network models alone are not enough to en-

able verification and automation; their variables need to also be populated with concrete values from the running network, and they should be continuously updated over time. Given the size and heterogeneity of modern networks, it can often be difficult in practice to gather and correctly interpret all the technical details about e.g. router configurations; instead, it may be needed to infer some parameters of the model (at a higher abstraction layer) based on other measurements. When measuring traffic in a large network, it is often necessary to perform sampling of the traffic and rely on statistical estimates based on the samples; moreover, with hundreds or even thousands of routers, it is hard to avoid gaps and errors in the measurements from each router. These are practical challenges related to mining the data from the network.

Specification: As the formal specification of desirable behavior for a computer network needs to express a variety of properties, including qualitative correctness and safety properties as well as quantitative performance properties, the formalisms developed to model the network need to be sufficiently expressive to model and check these properties. Moreover, the specification formalisms should be intuitive and easily understandable by non-experts to ensure clarity of the specification and coherence between the formal property and the intended behavior.

Optimization of configuration synthesis: While model verification can help avoid errors and increase the uptime of networks, the full benefit of formal methods for automating network management and improving network reliability and performance comes from being able to synthesize correct and efficient configurations from a formal specification. The challenge is to efficiently search through the huge state-space of possible configurations and find one that is both correct and performant.

Resilience: When a failure happens, the network's resiliency mechanisms kicks in and alters the forwarding behavior. The network then has a new data plane that may no longer satisfy the specification. To be able to give a priori guarantees, the verification and synthesis need to take all possible failures into account, hence significantly increasing the state-space. When considering that multiple failures may overlap, the number of possible failure scenarios becomes exponential. This calls for the development of efficient model checking algorithms. Further, to compute availability guarantees, the formal model and algorithms need to take failure probabilities into account.

Deployment: Solving the theoretical and algorithmic challenges is an important step for the research, but to become directly applicable in the networking industry, tools that implement the formal techniques need also be developed with usability in mind. Tools with excessive run-times can be impractical for the daily management of networks, so the efficiency of the verification and synthesis algorithms matters. For verification, a negative result should be accompanied with a counterexample that explains the property violation. Formal methods is not a one-off solution, they need to also address how to manage

and evolve the formal model and specification as the network and its applications and requirements grow. Finally, the integration of formal tools with existing network management systems and practices can help the incremental adoption of formal techniques.

This thesis, to different degrees, contributes to pushing the state-of-the-art of these challenges.

3 Overview of Computer Networks

Before diving into the research on network verification and synthesis, we provide an overview of relevant concepts in computer networking.

The internet is composed of many networks, each functioning as an autonomous system (AS) and each typically owned and controlled by a distinct entity. These networks connect and share routes with each other using the border gateway protocol (BGP) [137]. This thesis, and much of the related work, mainly focus on the mechanisms of a single network, i.e. inside a single AS, and the possible interactions with its neighbors, since this is the scope that can be fully observed and controlled by a network operator.

Such a computer network can be modelled as a graph of routers and links called the network topology. The software and hardware elements on each router can logically be split into the control plane, responsible for computing and installing paths, and the data plane, responsible for forwarding packets along the paths computed by the control plane.

Traditionally, networks rely on the internet protocol (IP) [130] to perform forwarding. Here the packet's destination IP address is matched against the router's forwarding table using longest prefix matching [63]. The intra-AS routing is computed using an interior gateway protocol (IGP) such as IS-IS [28, 76] or OSPF [120]. These are distributed algorithms for finding the shortest paths between each pair of routers given some configurable link weights.

Software defined networking (SDN) [97, 117] is an emerging technology that takes a different approach by centralizing the control plane. The data plane rules are then pushed out from the central controller to the individual routers through standardized interfaces like OpenFlow [117]. In this way, SDN allows operators to program their networks and create sophisticated routing solutions, such as centralized and frequently adjusted traffic engineering [72, 77], enforcement of security policies [29], and detection of DDoS flooding attacks [23].

Multiprotocol label switching (MPLS) [140] is a well-established networking technology typically deployed inside a single AS and used for e.g. enterprise wide-area networks (WANs) to simplify and generalize traffic engineering (TE) and virtual private network (VPN). In MPLS networks, packets are forwarded based on labels rather than network addresses. The labels encode

paths to network endpoints and are established by MPLS control plane protocols, where the two main protocols are label distribution protocol (LDP) [10] and resource reservation protocol with traffic engineering (RSVP-TE) [12].

By encapsulating the packet in a label header, MPLS simplifies the forwarding table lookup and allows for more explicit engineering of traffic paths. An MPLS packet can be further encapsulated by another label—effectively creating a stack of labels, where the top label is used to decide the forwarding. This allows tunneling packets through a specific path by pushing a label at the beginning of the path and popping the label at the last hop.

Segment Routing (SR) [53, 54] is an emerging source routing technology, where the ingress router can encapsulate the packet with a list of segments used for forwarding. SR works on top of either MPLS, using a stack of MPLS labels, or IPv6, using a special header with a list of IP addresses.

With the demand of large traffic volumes and the constraints of bandwidth capacity on individual links, networks may need to balance the traffic load across different paths through the network. Equal-cost multi-path routing (ECMP) [73] accomplishes this by splitting traffic among multiple equally short paths according to the link weights for the IGP.

When the failure of a link or a router is detected by the neighbors in the network, the control plane protocols start the distributed algorithms that recompute forwarding paths in the changed network topology. This is not a fast process, and in the meantime packets are lost. To mitigate this, networks can make use of fast reroute (FRR) [32, 133] techniques, where the router that detects the failure makes a local decision to switch the affected traffic onto a precomputed backup path. Compared to the reconvergence of the control plane algorithms, this local repair is very fast and hence reduces the number of packets that are lost due to the failure.

Internet service providers (ISPs) can specify guarantees on quality of service (QoS) [68] parameters such as bandwidth and latency as well as service availability and resilience through service level agreements (SLAs) with their clients. An SLA often impose a financial penalty on the ISP for breaking the contract [68].

4 Related Work

Over the past decade, academic research has taken an increasing interest into the formal treatment of computer networks. This section presents a selection of important literature in the area covering formalisms and tools for verification of both traditional and programmable network, optimization techniques for traffic engineering, and synthesis of network configurations and updates.

4.1 Verification of Traditional Networks

The seminal work in the field of network data plane verification by Xie et al. [170] proposes algorithms for static reachability analysis of IP networks. This is followed by work that extends static data plane analysis to more expressive properties and network models: ConfigChecker [4] and FlowChecker [3] uses binary decision diagrams (BDDs) [26] and CTL logic, Anteater [114] uses a SAT solver, and header space analysis (HSA) [91] models networks using bit-vectors and transformation functions. Real-time data plane analysis of rule updates is introduced using incremental algorithms in the tools Veriflow [92] and NetPlumber [90]. Further improvements to the scalability and expressiveness of real-time data plane analysis are achieved by Libra [178], Delta-net [74], the AP Verifier [171, 172], and APKeep [180].

More recently, P-Rex [45, 80] verifies MPLS network data planes and models the behavior of fast reroute for any scenario up to k link failures. P-Rex can handle the push and pop operations of MPLS networks without restricting the size of the label stack by using pushdown automata.

Configuration verification moves one abstraction layer up by taking all data planes that might emerge from the control plane routing protocols into account in the verification. Batfish [57] simulates the control plane and verifies properties using the SMT solver Z3 [40]. Batfish is able to parse configuration files from several network vendors. Its performance is improved by network-optimized datalog (NoD) [112] which extends Z3 Datalog with efficient data structures for modelling header spaces and packet rewrite functions. Minesweeper [14] extends Batfish with the ability to verify all failure scenarios with up to k failures. Bonsai [15] speeds up verification in Batfish and Minesweeper by compressing the control plane using abstraction refinement.

Several works attempt to improve the performance of configuration verification compared to Minesweeper. Both ARC [65] and ERA [50] use Batfish to parse configuration files and then perform efficient configuration verification of specific properties using respectively algorithms on weighted graphs and BDD operations. To simplify the use of Batfish as a frontend for configuration verifiers, NV [67] introduces an intermediate language for network configuration verification.

Tiramisu [2] creates a graph model of the network and uses different verification algorithms for different properties to balance performance and fidelity of the property, while Plankton [131] uses an executable model of the control plane protocols with the model checker SPIN [71]. Hoyan [174] combines simulation with formal verification in order to scale to the practical use in a real large network. Moreover, Hoyan is able to detect and deal with vendor-specific behavior in the running devices, hence addressing the correctness of the formal models used for verification. This is further addressed by Metha [18] that tests the correctness of network verifiers compared to the behavior of

real devices using fuzzing and combinatorial testing, and it uses delta debugging for fault localization. To scale network configuration verification further, Kirigami [161] and Timepiece [5] introduce modular control plane verification that allows parallel verification.

Real network configurations evolve gradually, so instead of performing a clean-slate verification on every configuration change, differential network analysis [179] uses techniques from differential dataflow programming and incremental verification to efficiently compute the impact in network behavior of a configuration change. To ease the adoption of network verification, Config2Spec [19] synthesizes formal network specifications from configurations.

Probabilistic network verification is introduced by NetDice [156] for verifying soft properties that must hold with a certain probability given a probabilistic failure model as a Bayesian network.

Stateful network verification [51, 126, 158, 176] addresses network verification in the presence of middleboxes, such as caches and firewalls, that change forwarding behavior based on the packets they observe. Given the high complexity of a priori verification of stateful network functions, Aragog [173] proposes to instead use runtime verification. Some verification approaches are specific to e.g. BGP [169] or datacenters [79, 129].

Much of the work in network verification focuses on qualitative correctness properties. It has recently been suggested to extend the use of formal methods to analyze quantitative network performance [11] and e.g. find counterexamples of traffic workloads that will cause bad performance.

4.2 Verification of Programmable Networks

Network programming is made possible by a standardization of the interfaces to proprietary routers and switches called OpenFlow [117]. On top of this, domain-specific SDN programming languages, like the Frenetic family [59, 60, 118, 119], give higher-level features for programming modular and portable network programs. The declarative language Merlin [155] allows provisioning network resources based on high-level policies and specified bandwidth constraints. Development of correct-by-construction SDN programs is addressed by Cocoon [141] using stepwise refinement.

A formal, semantic foundation for the programming of networks is introduced with NetKAT [9, 62] that builds on Kleene algebra with tests (KAT) [96]. NetKAT has a sound and complete equational theory that enables verification by checking the equivalence between safety properties and the programmed network behavior—both expressed in NetKAT. To reason about fault tolerance and expected congestion, ProbNetKAT [61, 153] extends NetKAT with probabilistic behavior, and McNetKAT [154] improves the verification performance by restricting to history-free probabilistic NetKAT programs. Another (history-free) probabilistic network programming language is Bayonet [64]

that builds on standard probabilistic programming and inference systems. WNetKAT [104] is a quantitative extension of NetKAT, but it lacks decision procedures. The recently proposed Concurrent NetKAT [166] extends NetKAT for reasoning about concurrency with multiple packets and mutable state.

While the first wave of SDN has allowed programming of a centralized control plane, switches and routers are now opened up to programmers with the data plane programming language P4 [21]. This allows implementing custom switch programs [152] or pushing advanced computations, such as performance diagnostics [66] or the Paxos consensus protocol [39], onto the network devices.

This new way of defining the forwarding behavior of networks has spawned a line of work on verification of P4 programs [111, 123, 157, 162]. The verification in Vera [157] uses symbolic execution, and p4v [111] is based on verification condition generation and SMT solving. Aquila [162] is practically used for verification of production-scale programmable data planes.

Recently, the development of formal foundations for the P4 language [43] has inspired formal reasoning tools for data plane programming based on interactive theorem proving [6, 44, 128, 167] and dependent type theory [46].

4.3 Traffic Engineering Optimization

Traffic engineering aims at steering the traffic demand efficiently through the network while satisfying the bandwidth capacity of each link and the latency requirements of the traffic. A classical approach for traffic engineering in IP networks [58] configures IGP link weights to minimize the maximum link utilization (relative to capacity).

More fine-grained control of the traffic paths and load balancing weights can be gained e.g. using RSVP-TE and weighted ECMP in MPLS networks or through the use of SDN. If moreover all the traffic demands are known, the maximum link utilization can be minimized by encoding it as a multi-commodity flow problem, which is well-studied and can be solved using linear programming (LP) [163]. In practice, traffic demands change over time and there might be scalability restrictions on the number of paths. To address the overhead involved in tearing down and setting up new paths when the traffic changes, SMORE [101] computes first a diverse set of low-stretch paths without knowledge of the traffic demands and then dynamically adapts sending rates based on the actual traffic.

Congestion-free resilience [85, 108, 168] is achieved by including the effect of failure protection in the optimization problems and guaranteeing congestion-freedom for certain numbers of failures. In some cases, congestion-free k-failure resilience is not possible, so more nuanced approaches [20, 31] use a probabilistic failure model to guarantee e.g. congestion-freedom for a certain availability target. This allows achieving high utilization while still being

congestion-free with high probability [20, 84].

Traffic engineering is being practically used to efficiently steer traffic between datacenters [72, 77] and even to optimize a large-scale internet backbone [98].

4.4 Synthesis of Network Configurations and Updates

Network configuration synthesis tools [1, 16, 17, 47, 48, 122, 135, 142, 160, 175] create correct-by-construction network configurations based on high-level policies or based on partial configurations with holes that need to be filled out correctly.

The introduction of SDN, where network programs dynamically send out updates to the routers, gave rise to a new issue: given that networks are distributed systems, there is no guarantee that the updates are installed simultaneously, so how do we ensure that packets are forwarded correctly while the update is happening? This is addressed by the notion of consistent network updates [136], which guarantees that each packet sees a consistent view of either the initial or final configuration. This strong consistency comes with an overhead of duplicating the forwarding tables and tagging packets during the update, so a relaxation is proposed [113] that allows other intermediate states but requires that they satisfy a set of safety properties.

McClurg et al. [115] and their tool, sometimes referred to as NetSynth, specifies these properties in LTL and uses incremental model checking to find safe sequences of rule updates. Not all network update problems can be solved by a sequences of rule replacements, so FLIP [165] generalizes the solution space to also contain rule additions in the update sequence and is able to solve more update problems.

Recent approaches using Petri nets [33, 41, 87] and Stackelberg games [146] benefit from existing formal models and efficient solvers and address the efficiency of the synthesized update schedules by finding safe batch updates. AllSynth [103] uses BDDs to find all update schedules that satisfy an LTL specification. Quantitative aspects are addressed by zUpdate [109] that synthesizes congestion-free network updates for datacenters and by Dionysus [86] that reduces update latency by dynamically scheduling the consistent updates based on the runtime differences of updating each switch. For updating the configurations of traditional routing protocols, SnowCap [147] finds reconfiguration orderings that satisfy an LTL formula of regular path conditions and among these it optimizes an objective function, such as minimizing the traffic shift during the safe update.

Foerster et al. [56] gives a comprehensive survey of consistent network updates.

5 Contributions

This thesis contributes to the formal treatment of network data planes with a focus on MPLS networks, where the forwarding is based on a label stack. For network verification, we build on the relation between MPLS networks and pushdown automata, where we address quantitative specifications, the scalability and efficiency of algorithms and implementations, as well as the trustworthiness of answers provided by verification tools. We improve the resilience of MPLS networks to multi-failure scenarios and synthesize failover protection that maintains certain properties of the data plane like loop-freedom. Finally, we address the challenge of obtaining an accurate model of a real network, in particular with respect to the ratios used when load balancing traffic among multiple paths.

The following conference publications are part of this thesis.

Paper A R-MPLS: Recursive Protection for Highly Dependable MPLS Networks. S. Schmid, M. K. Schou, J. Srba, and J. Vanerio. In: Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22), pp. 276–292, Association for Computing Machinery, 2022. [145]

This work led to a patent application (2022-521/10-0946) being filed on April 26, 2022 by Aalborg University in collaboration with the University of Vienna. The patent application was later discontinued.

Paper B MPLS-Kit: An MPLS Data Plane Toolkit. J. Vanerio, S. Schmid, M. K. Schou, and J. Srba. In: IEEE 11th International Conference on Cloud Networking (CloudNet '22), pp. 49–54, IEEE, 2022. [164]

(Presented at IEEE Global Internet (GI) Symposium 2022, organized in conjunction with IEEE CloudNet 2022.)

Paper C Faster Pushdown Reachability Analysis with Applications in Network Verification. P. G. Jensen, S. Schmid, M. K. Schou, J. Srba, J. Vanerio, and I. van Duijn. In: Automated Technology for Verification and Analysis (ATVA 2021), Lecture Notes in Computer Science, vol. 12971, pp. 170–186, Springer, 2021. [81]

Paper D AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks. P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba. In: Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), pp. 474–481, Association for Computing Machinery, 2020. [83]

This paper is a modified version of the master's thesis [99], compared to which a new experimental evaluation is performed and the clarity of presentation improved.

Paper E PDAAAL: A Library for Reachability Analysis of Weighted Pushdown Systems. P. G. Jensen, S. Schmid, M. K. Schou, and J. Srba. In: Automated Technology for Verification and Analysis (ATVA 2022), Lecture Notes in Computer Science, vol. 13505, pp. 225–230, Springer, 2022. [82]

Paper F Differential Testing of Pushdown Reachability with a Formally Verified Oracle. A. Schlichtkrull, M. K. Schou, J. Srba, and D. Traytel. In: Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD 2022), pp. 369–379, TU Wien Academic Press, 2022. [143]

Paper G Discovery of Flow Splitting Ratios in ISP Networks with Measurement Noise. M. K. Schou, I. Poese, J. Srba. Under submission, 17 pages (in this thesis), 2023.

In the following sections, we will present the contributions of the papers in this thesis—referring to them by their letters above. The papers are inserted in their entirety in Part II. In Section 6, we first introduce details of MPLS networks and present our formal model of these. We also describe the tool MPLS-Kit that is developed to aid the prototyping and evaluation of new ideas for MPLS networks, and we present one such idea, R-MPLS, for increasing the resilience of MPLS networks by synthesizing loop-free recursive failover protections. Section 7 motivates the use of pushdown automata for model checking MPLS networks and presents our algorithmic improvements as well as an extension for quantitative verification through the tools PDAAAL and AalWiNes. We increase the trust in results from these tools by formalizing the pushdown reachability algorithms in Isabelle/HOL and performing differential testing against this formally verified oracle. Our method for inferring flow splitting ratios from noisy measurements of a real network is described in Section 8.

6 MPLS Network Resilience

This section covers work from Paper A and Paper B on modelling, generating and analyzing MPLS data planes as well as synthesizing low-overhead, loop-free resiliency schemes for MPLS networks.

6.1 MPLS Network Model

As briefly discussed in Section 3, multiprotocol label switching (MPLS) [140] is a networking technology, where packets are forwarded based on labels rather than addresses. MPLS allows encapsulating the packet in a stack of labels, where each router forwards the packet based on the top label on the stack.

MPLS networks transport packets from an ingress to an egress router. When a packet enters the network, the ingress router classifies it and determines its forwarding equivalence class (FEC), which typically represents a

network resource, e.g. a specific router or a traffic engineering tunnel. The packets belonging to a FEC are forwarded in the same way through the network. The ingress router has a table that maps the FEC to an MPLS label. It encapsulates the packet in an MPLS header and initializes the stack with the corresponding label; the packet has now entered the MPLS data plane. In Paper B we provide a more detailed overview of MPLS networks.

When forwarding an MPLS packet, the router looks up the label on top of the stack in its forwarding table to find a matching forwarding entry. The forwarding entry contains a sequence of operations, instructing the router to modify the label stack, and the next-hop interface, where the packet should exit the router. Each operation acts on the top of the stack and can either swap, push, or pop a label.

In Figure 1 we introduce a small running example. Here Figure 1a depicts a network topology as a directed graph (to simplify the presentation some links are unidirectional), and Figure 1c lists the forwarding tables of the four routers. As an example of the forwarding process, consider a packet arriving at router v_1 with label stack $20 \circ 05$, where the left-most label denotes the top of the stack. A look-up in the forwarding table determines that the header should be modified by the operation $\texttt{swap}(21)$, resulting in the header $21 \circ 05$, and then the packet is sent out on link e_1 towards router v_2.

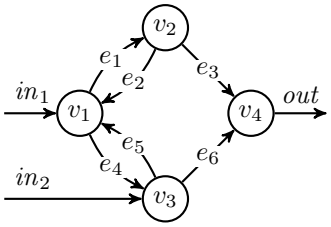
For the distribution of labels and installation of forwarding rules to encode desired traffic paths, as we describe in Paper B, the MPLS control plane has access to information about the network topology and shortest paths obtained by the IGP. We focus here on two main control plane protocols for MPLS:

- The label distribution protocol (LDP) [10] maps the IP address of each router and link to a locally recognized MPLS label and installs forwarding rules to reach this resource. It works by each router allocating local labels for each resource (FEC) it knows and then broadcasting this information to its neighbors. They will in turn allocate a local label for each of these resources, install the corresponding forwarding rules, and broadcast the new information to their neighbors.

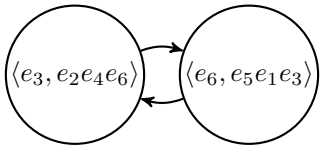
- The resource reservation protocol with traffic engineering extension (RSVP-TE) [12] can create label-switched paths (LSPs) from ingress to egress router. It is possible to assign traffic engineering policies on these paths by e.g. specifying certain waypoints, excluding some links from the path, or requiring some minimum bandwidth guarantee.

LDP creates basic connectivity in an MPLS network, but it does not allow the more fine-grained engineering of traffic paths that is possible with RSVP-TE.

MPLS networks can support balancing of the traffic load among multiple paths through the use of equal-cost multi-path routing (ECMP) [73]. For each packet, a forwarding rule is chosen pseudo-randomly; either uniformly or



(a) Network topology.



(b) Protection graph

Router	Label	Prio.	e_{out}	Operation
v_1	01	1	e_1	swap(02)
		1	e_4	swap(04)
	10	1	e_4	swap(11)
	20	1	e_1	swap(21)
v_2	02	1	e_3	swap(03)
		2	e_2	swap(03) ∘ push(10)
	21	1	e_3	pop
v_3	04	1	e_6	swap(05)
		2	e_5	swap(05) ∘ push(20)
	07	1	e_6	swap(08)
		2	e_5	swap(08) ∘ push(20)
	11	1	e_6	pop
v_4	03	1	out	pop
	05	1	out	pop
	08	1	out	pop

(c) Forwarding table

$(in_1, 01)(e_1, 02)(e_3, 03)(out, \varepsilon)$ $F = \emptyset$

$(in_1, 01)(e_1, 02)(e_2, 10 \circ 03)(e_4, 11 \circ 03)(e_6, 03)(out, \varepsilon)$ $F = \{e_3\}$

(d) Two traces through the network. The second uses FRR protection to route around a link failure.

Router	Label	Prio.	e_{out}	Operation
v_2	21	2	e_2	swap(10)
v_3	11	2	e_5	swap(20)

(e) Problematic forwarding rules introduce the possibility of a loop.

$(in_1, 01)(e_1, 02)\underline{(e_2, 10 \circ 03)}(e_4, 11 \circ 03)(e_5, 20 \circ 03)(e_1, 21 \circ 03)\underline{(e_2, 10 \circ 03)}\ldots \quad F = \{e_3, e_6\}$

(f) Looping trace (sequence of link-header pairs) in the failure scenario F using the rules from Figure 1e.

Fig. 1: Example MPLS network data plane with an LSP from v_3 to v_4 and load balancing between two LSPs from v_1 to v_4. Links e_3 and e_6 have FRR protections using labels (10, 11) resp. (20, 21).

based on a weight [88]. To avoid that packet reordering disrupts the performance of the transmission control protocol (TCP) on the transport layer, packets belonging to the same TCP flow should follow the same paths through the network. This is accomplishes by computing a hash of the header fields that identify the flow and using this as the key for selecting the forwarding rule [73].

To model ECMP, we allow the forwarding table to have multiple entries for a single label. In the running example in Figure 1c, this can be seen at router v_1, where the label 01 has two entries encoding the two different paths to v_4 going via either v_2 or v_3.

When we reason about the possible packet traces through the network, it is sufficient for the semantics to model that any of the possible entries can be chosen. When looking at certain quantitative properties of the data plane, the model needs to include a weight for each entry to indicate the relative probability that this entry is chosen. Paper G investigates how to obtain these

values in a real network analytics deployment.

Modelling Failures and Fast Reroute Protection

In the event of a failure of a router or a link, the neighboring routers will detect it and start notifying the rest of the network. This changed view of the network topology makes the control plane start recomputing shortest paths in the network. As mentioned in Section 3, this is a comparatively slow process, and while it unfolds, packets will be dropped at the point of failure. To combat this, networks can use fast reroute (FRR) [32] techniques to reduce the packet drop when a link or router fails by redirecting the traffic onto a precomputed backup path. The decision to redirect traffic happens locally at the router detecting that its corresponding interface port is down.

There are various techniques for failure detection, ranging from higher layer methods such as bidirectional forwarding detection [89] to lower layer techniques such as detecting a degradation in the signal quality of optical fibers e.g. observed by an increase in the bit error rate computed in the process of forward error correction [30, 75, 149]. We abstract over this by merely assuming the indication of whether an interface (and the corresponding link) is up or down.

In MPLS networks FRR is supported by RSVP-TE [125]. In its popular variant, facility protection, the protocol installs a backup tunnel around each facility, i.e. link or node, along the path to protect. In the case of a failure, the repairing router pushes the label that encodes the backup tunnel onto the label stack and the packet is forwarded along the protection path. At the last hop, the label is popped, and the packet merges back in to its original path, using the underlying label for forwarding.

We model FRR mechanisms in the MPLS data plane by assigning a priority to each forwarding table entry. The semantics is now to find the entries with highest priority among those where the next-hop interface is up. If multiple entries with equal priority qualify, the semantics for ECMP apply.

Returning to the running example, we see in Figure 1c that if the link e_3 is down, packets arriving at router v_2 with the label 02 on top of the stack will have to use the lower priority backup rule that pushes the label 10 on the stack and sends the packet along the backup path through links e_2, e_4, and e_6. Figure 1d shows first a packet trace on the primary path through router v_2 in the situation with no failures and then a trace where FRR protects against the failure of link e_3.

> **Contribution 1 (Paper A)**
> We present a formal model of MPLS data planes that is capable of modelling all possible packet traces through the network, including the behavior of ECMP and fast reroute in case of failures.

6.2 Data Plane Generation and Simulation

When performing research of new networking techniques and protocols, it is imperative to be able to experiment with the new ideas on realistic test cases to assess performance and to possibly discover unseen issues. While there exist datasets of network topologies (the graph of routers and links) stemming from measurements on real-world networks [95], there is a lack of open source datasets of network data planes and control planes, in particular datasets that include the routers' configurations and forwarding tables.

Generating arbitrary forwarding tables is likely to result in data planes that differ widely from real networks, hence making research results based on these data planes inaccurate compared to a real deployment. To enable research into MPLS networks, including the evaluation of formal verification and synthesis tools, we develop in Paper B the automated tool MPLS-Kit that can generate realistic MPLS data planes based on a parameterized configuration using widely-deployed industry-standard control protocols.

MPLS-Kit is implemented as a modular Python library with a command line interface. It allows specifying which of the supported control plane protocols to use, including LDP, RSVP-TE with fast reroute, and VPN services, as well as the parameters for each protocol. These parameters can for instance include a list of specific traffic engineering tunnels or just a number of random tunnels to create. This flexibility allows both simple and advanced uses.

Given the specified configuration, MPLS-Kit produces an MPLS data plane that follows the model presented in the previous section. To efficiently generate the data plane, the tool abstracts over certain details of the distributed control-plane algorithms and produces the data plane that would result from the convergence of this distributed process.

On the generated data plane, MPLS-Kit can perform simple packet-level simulations, from which packet traces and various statistics can be gathered. In Paper B we present use cases of MPLS-Kit's simulation including congestion, latency, and resilience analysis. The modular and extensible design allows easy prototyping of new techniques, such as the R-MPLS protection scheme that we will introduce in the next section.

> **Contribution 2 (Paper B)**
> We present MPLS-Kit, an open-source tool for automated and efficient generation and simulation of synthetic, yet realistic, MPLS data planes based on parameterized configurations.

6.3 Recursive Fast Reroute Protection

While the fast reroute supported in MPLS networks by RSVP-TE can protect against single link or node failures, it does not provide resilience to multiple concurrent failures. The FRR technique installs backup tunnels around vulnerable links or nodes to protect the main traffic, but it does not attempt to protect against failures in the backup paths.

Given that most links and nodes already have protection paths around them for protecting the main traffic paths, it seems like a low-hanging fruit to increase the resilience by reusing these paths to protect the protection paths—resulting in recursive protection. Since applying a facility protection amounts to just pushing a corresponding label onto the stack, recursive facility protection is a fast and memory efficient way of providing increased resilience against multi-failure scenarios.

To understand why existing FRR does not perform this recursive protection, consider the two rules in Figure 1e that "protect" the backup path $e_5e_1e_3$ against of failure of link e_3 and similarly for backup path $e_2e_4e_6$ and the link e_6. If both e_3 and e_6 are down, e.g. caused by a node failure on v_4, the recursive protection will keep sending packets back and forth between routers v_2 and v_3 until the time-to-live (TTL) field of the packets reach zero. This forwarding loop will quickly exhaust the bandwidth of the impacted links (e_1, e_2, e_4 and e_5), which effectively blocks traffic that would otherwise flow unhindered on other LSPs from v_2 to v_3 via v_1.

As this example highlights, naïvely applying recursive protection can result in packets being forwarded in a loop—leading to detrimental performance. We will now look at how to proactively avoid this and develop the R-MPLS technique (presented in Paper A) that guarantees loop-free recursive protection. In the example, we can simply avoid installing the two rules in Figure 1e, but in general we want to install as many recursive protections as possible while guaranteeing that no forwarding loops are introduced. The challenge is to determine which recursive protections are safe to install.

To capture the possible interplay of different protection paths, we define a graph, where each node is a protection, modelled as a pair of the failing link and the sequence of links used as the protection path. This protection graph has a directed edge from p to q if the protection q can be used for recursive protection somewhere along the path of the protection p. The idea is that a cycle in this graph corresponds to a possible forwarding loop in the network under some failure scenario if all these recursive protections are enabled. Paper A presents the formal definition and further uses certain annotations of the directed edges in the graph to give a precise characterization of which cycles in the graph correspond to forwarding loops in the network—details that we will omit in this overview.

The protection graph for the two protections in the running example, de-

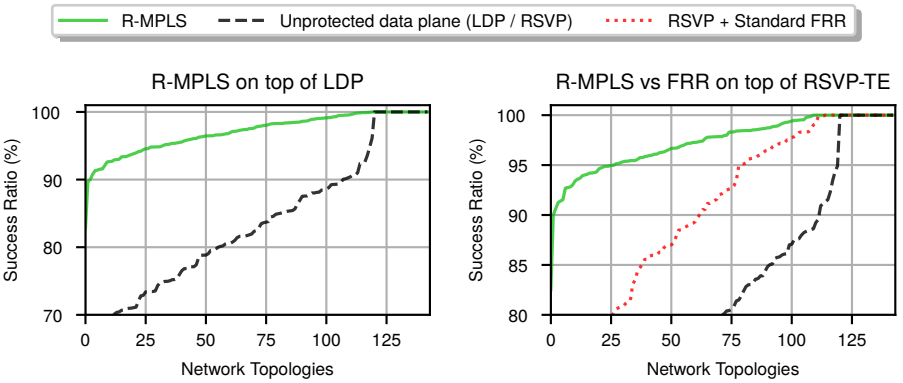


Fig. 2: Success rate per topology relative to optimal protection (higher value is better) for R-MPLS on top of LDP (left) and RSVP-TE (right) compared to the unprotected data plane. For RSVP-TE we also compare to its standard FRR protection. Plots are from Paper A.

picted in Figure 1b, has a cycle that corresponds to the forwarding loop in the scenario where links e_3 and e_6 fail as shown by the trace in Figure 1f.

The loop-avoidance algorithm proceeds by finding cycles in the protection graph and removing edges until there are no more cycles that correspond to forwarding loops in the network. When removing an edge (p, q) from the graph, we add it to a set of disabled protection pairs and make sure that R-MPLS does not install the recursive protection q for the label that encodes the protection p. This is how the algorithm determines that the rules in Figure 1e should not be installed, as they correspond to the two edges forming a cycle in Figure 1b. In Paper A we prove that R-MPLS, using this loop-avoidance algorithm, is guaranteed not to introduce any forwarding loops.

Contribution 3 (Paper A)
We present R-MPLS, a recursive fast reroute protection scheme for practical multi-failure resilience in MPLS networks. R-MPLS uses an algorithm for proactively avoiding any loops that would otherwise result from recursive application of backup paths. We prove that R-MPLS with this loop-avoidance algorithm guarantees loop-free protection.

To evaluate R-MPLS, we use MPLS-Kit (Paper B) to generate data planes on 143 network topologies from the internet topology zoo dataset [95] using either LDP or RSVP-TE to populate the forwarding tables. On top of these baseline data planes we run R-MPLS to create recursive protections, and for RSVP-TE we run its standard FRR with facility protection for comparison.

Next, we enumerate failure scenarios of up to four link failures for each data

plane, and we run packet-level simulations to determine how many packets the data plane successfully forwards in the given failure scenario. Packets that get physically disconnected from their destination by the failure scenario are not counted, so a theoretical optimal protection achieves 100% success rate. In Figure 2 we plot the success rate averaged over all simulated failure scenarios, where for each curve the topologies are sorted by their success ratio. On the LDP-based data planes, we see that R-MPLS significantly improves the resilience compared to the unprotected data plane. The same trend is clear on the RSVP-TE data planes, where the recursive protection also improves greatly on the standard FRR that only protects against single failures.

These experiments are presented in more detail in Paper A along with an analysis showing that R-MPLS induces only a modest overhead in memory (needed for storing forwarding rules) and communication (between routers needed for the distributed computation of forwarding tables).

Contribution 4 (Paper A)
We show through extensive experiments that R-MPLS achieves high resilience with only a modest memory and communication overhead. R-MPLS builds upon standard protocols and can be implemented in a distributed way with existing MPLS hardware.

7 Pushdown Automata Reachability

In this section, we shall introduce our contributions to automated formal verification of MPLS networks. The work arises from the observation [80, 144] that the packet forwarding of an MPLS network, with a label stack modified by push, pop, and swap operations, closely resembles the classical formal model of a pushdown automaton [151, Chapter 2.2].

Our work extends on a research prototype tool for MPLS network verification, P-Rex [80], by adding quantitative verification; further, we propose and implement several efficiency improvements that combined result in orders of magnitude speed up, making the MPLS network verification more scalable for practical use.

7.1 Pushdown Automata Reachability for MPLS Verification

A pushdown automaton (PDA) is a finite-state automaton equipped with an unbounded stack and rules that can change state and modify the stack if they match the top symbol on the current stack. Formally, a PDA has a finite set of control locations P, a finite input alphabet Σ, a finite stack alphabet Γ, and a

finite set of rules on the form $\langle p, \gamma \rangle \overset{\sigma}{\hookrightarrow} \langle p', w \rangle$ where $p, p' \in P$, $\gamma \in \Gamma$, $w \in \Gamma^*$, and $\sigma \in \Sigma$. (Note the use of Kleene star means that Γ^* is the set of all words over the alphabet Γ, i.e. sequences of elements from Γ; moreover, let ε denote the empty word.) The rule says that in control location p, if γ is on top of the stack, we can move to p' and replace γ with w.

A configuration in a PDA is a pair $\langle p, w \rangle$ of a control location $p \in P$ and a stack $w \in \Gamma^*$. A trace in the PDA is a sequence of Σ-labelled transitions between configurations $\langle p_0, w_0 \rangle \overset{\sigma_1}{\Longrightarrow} \langle p_1, w_1 \rangle \overset{\sigma_2}{\Longrightarrow} \ldots \overset{\sigma_n}{\Longrightarrow} \langle p_n, w_n \rangle$ such that for each step $\langle p_i, w_i \rangle \overset{\sigma_{i+1}}{\Longrightarrow} \langle p_{i+1}, w_{i+1} \rangle$ there is a rule $\langle p_i, \gamma \rangle \overset{\sigma_{i+1}}{\hookrightarrow} \langle p_{i+1}, w \rangle$ and a suffix $w' \in \Gamma^*$ such that $w_i = \gamma w'$ and $w_{i+1} = w w'$. Note how the rule must match the current control location and top-of-stack symbol, and that everything in the stack below the top symbol is unchanged when taking a step. We sometimes omit the Σ-labels when just talking about the existence of a trace[1].

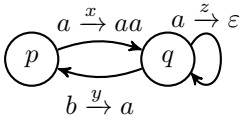
For example, in a PDA with $P = \{p, q\}$, $\Sigma = \{x, y, z\}$, $\Gamma = \{a, b\}$, and the three rules $\langle p, a \rangle \overset{x}{\hookrightarrow} \langle q, aa \rangle$, $\langle q, b \rangle \overset{y}{\hookrightarrow} \langle p, a \rangle$, and $\langle q, a \rangle \overset{z}{\hookrightarrow} \langle q, \varepsilon \rangle$, a possible trace is $\langle p, ab \rangle \overset{x}{\Rightarrow} \langle q, aab \rangle \overset{z}{\Rightarrow} \langle q, ab \rangle \overset{z}{\Rightarrow} \langle q, b \rangle \overset{y}{\Rightarrow} \langle p, a \rangle$. Notice how the three rules correspond respectively to push, swap, and pop operations. This PDA is depicted in Figure 3a and will be used as a running example.

For MPLS network verification, we let Σ be the set of (directed) links in the network and Γ be the set of MPLS labels. This allows us to model both the links traversed by a packet and its intermediate MPLS headers. In a failure-free scenario, the translation from our MPLS data plane model to pushdown automata follows straightforwardly—details of this translation are presented in Paper C. We will later discuss how to handle failures and fast reroute, i.e. lower priority rules in the MPLS data plane model.

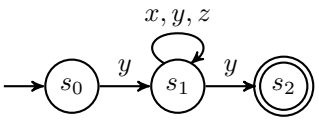
Note that this presentation of pushdown automata omits the start configuration and accepting control locations that are normally part of a PDA [151], since there is no clear way to define the start and end of an MPLS network. This means that we cannot talk about the language accepted by the PDA modelling the network; instead, we will use a query language to specify a subset of traces that are of interest.

This query language, which was first introduced by P-Rex [80], specifies network traces by means of an expression on the form $\langle regex_I \rangle\ regex_{path}\ \langle regex_F \rangle$, where $regex_{path}$ is a regular expression over network links, describing traversed paths, while $regex_I$ and $regex_F$ are regular expressions over labels, describing the initial resp. final header of the network trace. This gives a way of expressing traces with various properties in an intuitive language. We formalize this as the following problem:

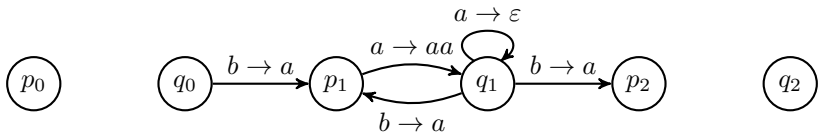
[1]Some papers [22, 81, 82, 138, 143, 148] (including Paper C, Paper E, and Paper F) call such a structure a pushdown system (PDS).



(a) PDA $\mathcal{P}_1$ where rules are arrows between control locations annotated with stack modification and rule label



(b) NFA $\mathcal{A}_1$ for the regular expression $y\,(x|y|z)^*\,y$



(c) PDA $\mathcal{P}_2$ as the product of $\mathcal{P}_1$ in (a) and $\mathcal{A}_1$ in (b). E.g. q_0 corresponds to q and s_0. Rule labels omitted.

$$\langle q_0, bb\rangle \Rightarrow \langle p_1, ab\rangle \Rightarrow \langle q_1, aab\rangle \Rightarrow \langle q_1, ab\rangle \Rightarrow \langle q_1, b\rangle \Rightarrow \langle p_2, a\rangle \quad (1)$$

$$\langle q, bb\rangle \overset{y}{\Rightarrow} \langle p, ab\rangle \overset{x}{\Rightarrow} \langle q, aab\rangle \overset{z}{\Rightarrow} \langle q, ab\rangle \overset{z}{\Rightarrow} \langle q, b\rangle \overset{y}{\Rightarrow} \langle p, a\rangle \quad (2)$$

$$s_0 \xrightarrow{y} s_1 \xrightarrow{x} s_1 \xrightarrow{z} s_1 \xrightarrow{z} s_1 \xrightarrow{y} s_2 \quad (3)$$

(d) A trace (1) through $\mathcal{P}_2$ in (c), and corresponding labelled trace (2) in $\mathcal{P}_1$, and path (3) in $\mathcal{A}_1$

Fig. 3: Example of a PDA $\mathcal{P}_1$, NFA $\mathcal{A}_1$, and their product construction into a new PDA $\mathcal{P}_2$

Problem 1 (Pushdown Query Satisfiability). Given a PDA $\mathcal{P}$ and a query $\langle regex_I\rangle\ regex_{path}\ \langle regex_F\rangle$, return, if it exists, a trace $\langle p_0, w_0\rangle \overset{\sigma_1}{\Longrightarrow} \ldots \overset{\sigma_n}{\Longrightarrow} \langle p_n, w_n\rangle$ in $\mathcal{P}$ such that $\sigma_1 \ldots \sigma_n$ matches the regular expression $regex_{path}$, w_0 matches $regex_I$, and w_n matches $regex_F$.

For the PDA in the running example (see Figure 3a), say for instance we wish to find traces that start and end with an edge labelled y, the traces can have any initial header, and the final header must be the single label 'a'. We can express this as regular expressions with the query $\langle\, \cdot^*\rangle\ y\ \cdot^* y\ \langle\, a\,\rangle$, where the symbol '$\cdot$' is a wildcard that matches any symbol in the corresponding alphabet. It is well known that any regular expression can be transformed into an equivalent finite automaton [151].

A nondeterministic finite automaton (NFA) consists of a finite set of states S with some states designated as initial $I \subseteq S$ and final $F \subseteq S$, an alphabet, and a transition relation on the form $s \xrightarrow{\sigma} s'$, where $s, s' \in S$ and σ is in the alphabet. For the path expression in the query, the alphabet is Σ; while for specifying the headers, the alphabets of the NFA are Γ. A word $w = \sigma_1 \ldots \sigma_n$ is in the language of the NFA, if there are states $s_0, \ldots, s_n \in S$ such that $s_0 \in I$, $s_n \in F$, and $s_i \xrightarrow{\sigma_{i+1}} s_{i+1}$ for $0 \leq i < n$. We may also write this as $s_0 \xrightarrow{w}{}^* s_n$ and say that w is accepted by the automaton. A set of words is said to be regular if it is the language of a finite automaton.

For the running example, Figure 3b shows an NFA for the regular expression $y \cdot^* y$, where s_0 is the initial state and s_2 is the final state. The word $yxyy$

is accepted by this NFA, while the word $xzzy$ is not.

If a trace in a given PDA $\mathcal{P}_{network}$ satisfies a query $\langle regex_I \rangle\ regex_{path}\ \langle regex_F \rangle$, the labels of the rules in the trace must form a word that is recognized by the path expression $regex_{path}$ of the query and the NFA $\mathcal{A}_{path}$ that encodes it. To find such traces, we combine $\mathcal{P}_{network}$ and $\mathcal{A}_{path}$ into a new PDA $\mathcal{P}_{product}$ by means of a product construction, where each control location in $\mathcal{P}_{product}$ is a pair of a control location from $\mathcal{P}_{network}$ and a state from $\mathcal{A}_{path}$. The stack alphabet of $\mathcal{P}_{product}$ is the same as in $\mathcal{P}_{network}$, and the rules of $\mathcal{P}_{product}$ are constructed so that for each rule $\langle p, \gamma \rangle \overset{\sigma}{\hookrightarrow} \langle p', w \rangle$ in $\mathcal{P}_{network}$ and transition $s \xrightarrow{\sigma} s'$ in $\mathcal{A}_{path}$, where the symbol σ matches, the rule $\langle (p, s), \gamma \rangle \overset{\sigma}{\hookrightarrow} \langle (p', s'), w \rangle$ is added to $\mathcal{P}_{product}$.

Figure 3c shows, for the running example, the result $\mathcal{P}_2$ of the product construction of the PDA $\mathcal{P}_1$ in Figure 3a and the NFA $\mathcal{A}_1$ in Figure 3b. The names of the control locations in $\mathcal{P}_2$ are simplified so that e.g. p_1 is the pair (p, s_1) of corresponding control location and state from $\mathcal{P}_1$ and $\mathcal{A}_1$. Any trace in $\mathcal{P}_2$ corresponds to a trace in $\mathcal{P}_1$ and a path in $\mathcal{A}_1$ with the same Σ-labels. If the trace starts in either p_0 or q_0 and ends in either p_2 or q_2, the corresponding path in the NFA will be an accepting path. Figure 3d shows an example of such a trace.

Next, to handle the query's regular expressions on the initial and final headers, we translate them into NFA over the stack alphabet Γ. We extend each of these NFA from specifying a set of stacks to specifying a set of PDA configurations by letting the control locations of the PDA become initial states in the NFA. We say that a configuration $\langle p, w \rangle$ is accepted by this NFA, if $p \xrightarrow{w}{}^* q$ for some final state q in the automaton. This allows us to specify both control locations and stacks in the same structure, and this special NFA turns out to be a natural fit for the reachability algorithms used to find traces in the PDA [148], which helps make this construction efficient.

As shown in Paper C, we have now reduced Problem 1 of finding a PDA trace that satisfies a given query to the following pushdown reachability problem:

Problem 2 (Pushdown reachability). Given a PDA and two finite automata $\mathcal{A}$ and $\mathcal{A}'$, return, if it exist, a trace $\langle p_0, w_0 \rangle \Rightarrow \cdots \Rightarrow \langle p_n, w_n \rangle$ in the PDA such that $\langle p_0, w_0 \rangle$ is accepted by $\mathcal{A}$ and $\langle p_n, w_n \rangle$ is accepted by $\mathcal{A}'$.

For the running example, we create an initial NFA $\mathcal{A}$ that accepts $\langle p_0, w \rangle$ and $\langle q_0, w \rangle$ for any $w \in \{a, b\}^*$ and a final NFA $\mathcal{A}'$ that accepts only $\langle p_2, a \rangle$ and $\langle q_2, a \rangle$. Trace (1) in Figure 3d is an example of a solution to this instance of the pushdown reachability problem, and trace (2) in Figure 3d is the corresponding PDA trace that satisfies the query $\langle\ \cdot^*\rangle\ y\ \cdot^* y\ \langle\ a\ \rangle$.

Modelling Link Failures by Over-Approximation

The presentation so far has shown how to model MPLS networks in the failure-free scenario as a pushdown automata and how to specify traces using regular expressions. The last part of the query language from P-Rex [80] is to specify a maximal number of link failures k to consider when modelling the behavior of fast reroute protections. The semantics of the query is to ask whether there exists a failure scenario with at most k failures, in which there is a possible network trace that matches the regular expressions in the query.

The possibility of failures is handled by over-approximation: at each router, we include (in the creation of the network PDA) all forwarding rules that are active, i.e. may be used, in a failure scenario with at most k failures [80]. As described above, finding a trace that satisfies the three regular expressions in the query can be reduced to the pushdown reachability problem. Due to the over-approximation, when a satisfying trace is found, we need to check that no more than k links were assumed to be failed for the forwarding entries in the trace to activate and that each link was consistently assumed to be either active or failed.

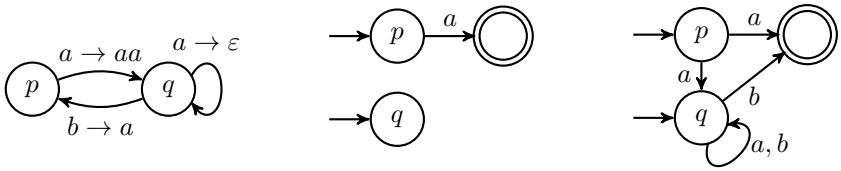
The over-approximation of failures and the idea of using a product construction of the network PDA and the NFA that encodes the path expression were already part of the solution in P-Rex [80]. The new idea in this translation from MPLS network verification to pushdown reachability is the direct use of finite automata to encode the initial and final headers and to constrain the trace to start and end with the initial resp. final states of the path NFA. This reduces the size of the pushdown automata compared to the approach in P-Rex [80], and together with an efficient C++ implementation of the translation it leads to more efficient verification.

> **Contribution 5 (Paper C)**
> We present a direct, efficient translation from the query satisfaction problem for MPLS network into the pushdown reachability problem.

7.2 Solving Pushdown Reachability Efficiently

After having established the use of pushdown reachability for MPLS network verification, this section will introduce our improvements to the algorithms for solving pushdown reachability. First, we will make a brief overview of the existing algorithms.

For a set of PDA configurations C, the predecessors of C are all configurations that have a trace to a configuration in C, and the successors of C are all configuration that are reachable by a trace from a configuration in C.



(a) PDA $\mathcal{P}_1$ from Figure 3a **(b)** NFA $\mathcal{A}'$ accepting $\{\langle p, a\rangle\}$ **(c)** Saturated NFA $\mathcal{A}'_{pre^*}$

Fig. 4: Example of applying the pre^* saturation procedure for PDA $\mathcal{P}_1$ on the automaton $\mathcal{A}'$

More formally, this is respectively $pre^*(C) = \{c \mid \exists c' \in C.\ c \Rightarrow^* c'\}$ and $post^*(C) = \{c' \mid \exists c \in C.\ c \Rightarrow^* c'\}$, where $c \Rightarrow^* c'$ indicates the existence of a trace (with zero or more steps) from c to c'.

The key insight for solving pushdown reachability is that if the set of configurations C is regular, then both $pre^*(C)$ and $post^*(C)$ are regular too [27]; moreover, the corresponding automata can be computed efficiently by adding transitions to the finite automaton representing C according to certain saturation rules [22, 55, 148]. For instance, pre^* works by the following saturation rule: if the PDA has a rule $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$, and in the current automaton $p' \xrightarrow{w}{}^* q$ for some state q, then add the transition $p \xrightarrow{\gamma} q$ to the automaton [148]. We will call these algorithms, specifically the versions presented by Schwoon [148], pre^* and $post^*$.

Figure 4 shows, for the PDA in the running example, the result of applying the pre^* algorithm on an automaton $\mathcal{A}'$ that accepts just the configuration $\langle p, a\rangle$. The resulting NFA, $\mathcal{A}'_{pre^*}$, in Figure 4c accepts $pre^*(\{\langle p, a\rangle\}) = \{\langle p, a\rangle\} \cup \{\langle p, awb\rangle \mid w \in \Gamma^*\} \cup \{\langle q, wb\rangle \mid w \in \Gamma^*\}$.

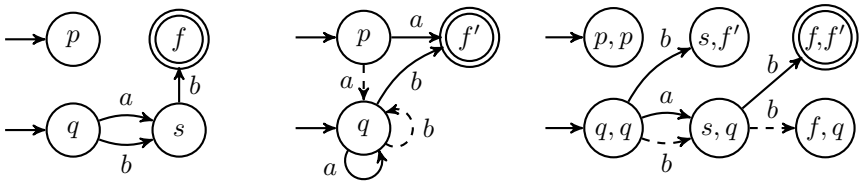
The next step in solving the pushdown reachability problem from a set of initial configurations C to a set of final configurations C', is to check if C intersects with $pre^*(C')$ or, symmetrically, if C' intersects with $post^*(C)$. Given that these sets are represented by finite automata, we can use a textbook product construction for the intersection of two finite automata [151].

In Figure 5 we see the product automaton for the intersection of $\mathcal{A}'_{pre^*}$ and an NFA $\mathcal{A}$ that accepts $\langle q, ab\rangle$ and $\langle q, bb\rangle$. We see that the languages of these two automata do intersect.

Finally, if reachability is satisfied, we need to return a trace as a witness of reachability. This can be achieved by annotating the transitions added by pre^* or $post^*$ with metadata that explain why the transition was added [148]. We can then backtrack from a configuration in the intersection and use the metadata to discover the rule applied at each step.

On-The-Fly Construction of Product Automaton and Bidirectional Search

The first improvement to the algorithms is to terminate as soon as reachability is determined to be satisfied. This is achieved by constructing on the fly the



(a) NFA $\mathcal{A}$ accepts $\{\langle q, ab\rangle, \langle q, bb\rangle\}$ **(b)** NFA $\mathcal{A}'_{pre^*}$ from Figure 4c **(c)** Product automaton of $\mathcal{A}$ and $\mathcal{A}'_{pre^*}$

Fig. 5: With on-the-fly construction of the product automaton of $\mathcal{A}$ and $\mathcal{A}'_{pre^*}$, the algorithm can terminate early before adding the dashed arrows to the automata.

product of the two finite automata, encoding C and $pre^*(C')$ or encoding $post^*(C)$ and C', while the pre^* or $post^*$ saturation algorithm adds transitions to the corresponding automata. The algorithm will early terminate when a non-empty intersection is found.

In the running example with the pre^* algorithm, only the transitions marked with solid arrows in Figure 5 have been added when an accepting path is found in the product automaton in Figure 5c. The algorithm can terminate early without having to add the two transitions with dashed arrows in Figure 5b that are part of the fully saturated pre^* automaton. For larger examples, this early termination can lead to a significant speed-up.

The idea behind the algorithm for on-the-fly construction of the product automaton, which we present in Paper C, is to keep track of states in the product automaton that are reachable from an initial state, and when a transition is added to one of the automata, we efficiently check if there are any matching transitions in the other automaton that can lead to the addition of a transition from a reachable state in the product automaton. When a new state becomes reachable, we recursively check if transitions from this state can be added. If a final state is reached in the product automaton, we know that we have found an accepting path, and the algorithm can terminate.

To evaluate the algorithms, we conduct benchmarks on a large set of MPLS network verification problems. Figure 6 shows the results, where we see that pre^* and $post^*$ have similar performance. In Paper C we show how this is already an improvement compared to previous implementations. Interestingly, the two algorithms often perform well on different problem instances (not shown in the plot), which spark the idea of combining the forward search of $post^*$ with the backward search of pre^*.

This leads us to our second algorithmic improvement for the pushdown reachability problem: bidirectional search, which we in Paper C name $dual^*$. The idea with $dual^*$ is to apply pre^* on the set of final configurations C' and $post^*$ on the set of initial configurations C in a way that interleaves the steps of the saturation procedures. We use the on-the-fly product construction on the corresponding two automata, where pre^* adds transitions to one automaton

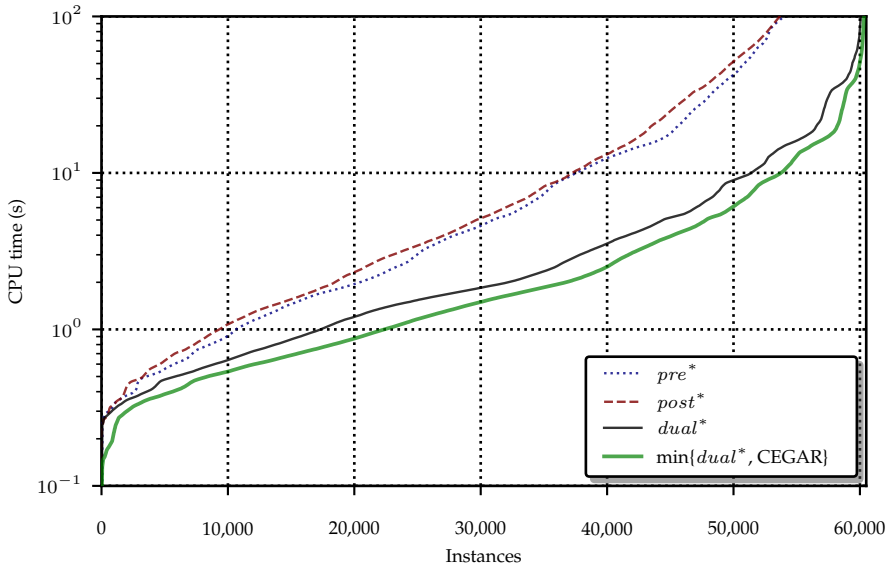


Fig. 6: Benchmark comparison showing the speed-up of $dual^*$ and its combination with CEGAR compared to pre^* and $post^*$. For each solver, all 60 800 instances on the x-axis are independently sorted by the verification time on the logarithmically scaled y-axis. Plot is from Paper C.

and $post^*$ adds transitions to the other. As soon as a configuration is found in the intersection between $post^*(C)$ and $pre^*(C')$, we know that there is a trace from C through this configuration to C', and the algorithm can terminate. Moreover, we can terminate the algorithm with a negative answer as soon as one search saturates without finding an overlap. This means that for unreachable cases, only the smallest of the two search spaces, $post^*(C)$ and $pre^*(C')$, needs to be exhausted. In Figure 6 we see how the $dual^*$ algorithm significantly improves the running time compared to the already optimized pre^* and $post^*$ implementations. Paper C gives the full algorithm.

Our third improvement is the implementation of counterexample-guided abstraction refinement (CEGAR) [34] for pushdown reachability. The idea with CEGAR is to reduce the size of the verified PDA by performing an abstraction [35] that groups control locations and labels into a smaller number of abstract control locations and labels. If a trace is found in this smaller PDA, we efficiently try to reconstruct a corresponding trace in the original PDA using the method described in Paper C. If this fails, the spurious trace is used as a counterexample to refine the abstraction so that the spurious trace is not present in the refined PDA in the next iteration. In some cases this makes verification much faster, since we are verifying a smaller model; however, in other cases the abstraction needs to be refined too many times—leading to a

longer overall verification time. For this reason, we suggest running CEGAR in parallel with the $dual^*$ algorithm to benefit from the cases where CEGAR is fast without suffering from its bad cases. As we see in Figure 6, this improves the running time even further and leads to almost an order of magnitude speed up compared to pre^* and $post^*$ on the large instances.

Contribution 6 (Paper C)
We improve the running time of pushdown automata verification by i) an on-the-fly algorithm for finite automata intersection, ii) a novel bidirectional search algorithm for reachability of pushdown automata, and iii) the use of counter-example guided abstraction refinement to decrease the size of the pushdown automaton. We demonstrate the performance improvements by benchmarking on a large set of network verification problems.

7.3 Weighted Pushdown Automata Verification

While it can be useful to know whether there exists a trace in an MPLS network with a given regular pattern, in the positive cases, it might not be enough to just have an arbitrary example of such a trace. For debugging the network, we may want to find a trace in a failure scenario with as few failures as possible, or for traffic engineering we can ask what the worst-case latency from A to B is when at most two links fail. To answer such questions, we introduce in Paper D quantitative what-if verification for MPLS networks with the tool AalWiNes.

Specifically, AalWiNes supports assigning weights to each forwarding rule in the network data plane and finding a query-satisfying trace that has either the smallest or the largest sum over these weights. The weights can encode e.g. hop-count, link latency, number of push operations, or the number of failures needed locally for the forwarding rule to be active. To allow expressing more advanced weight properties, AalWiNes supports linear combinations and lexicographically ordered vectors of such weights.

The theoretical foundation of this quantitative verification is weighted pushdown automaton (WPDA). In general, a WPDA is a pushdown automaton where each rule is assigned a weight. In order to analyze the weight of traces in a WPDA, the weights must come from a domain that supports two operations and satisfies certain properties like associativity and commutativity. Specifically, the domain must define an operation for computing the weight of a trace from the weights of its steps, and it must define an operation for aggregating over the weights of multiple traces. Formally, we use an algebraic structure known as an idempotent semiring for the weighted generalization of pushdown reachability [82, 100, 138]. In AalWiNes and its underlying PDA library PDAAAL (presented in Paper E), we require the weight domain to

Fig. 7: The graphical user interface of AalWiNes showing the result of a longest trace query on a small example network.

be totally ordered. Other works on WPDA use different relaxations of this requirement [100, 138].

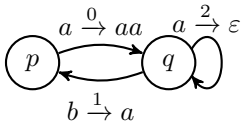
The domains we use for quantitative MPLS verification are integers and vectors of integers. The weight of the trace is the (elementwise) sum of the weights of rules on the trace, and we aggregate over multiple paths by the (lexicographical) minimum or maximum depending on the whether we are looking for shortest or longest traces.

To improve usability, AalWiNes features a graphical user interface, shown in Figure 7, for inspecting the network model, specifying queries and weight expressions, and visualizing the returned traces.

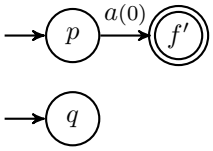
> **Contribution 7 (Paper D)**
> We extend previous work on MPLS network verification to include quantitative verification. In particular we implement an efficient tool, AalWiNes, that supports shortest and longest trace queries using various configurable weights such as hop-count or latency.

For solving the (weighted) pushdown reachability problems from AalWiNes, we build in Paper E an efficient open-source C++ library and command-line tool for (weighted) pushdown automata verification, PDAAAL.

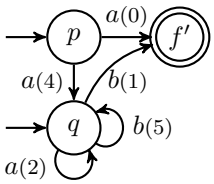
The idea behind extending the pre^* and $post^*$ algorithms for pushdown reachability to finding e.g. shortest traces in WPDA, is that the saturation procedures not only add transitions to the NFA, they also assign weights to the transitions in a way so that the weight of a shortest accepting path for a given



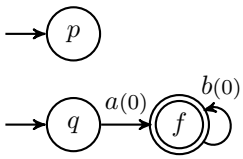
(a) WPDA modified from Figure 3a with weight annotations on rules



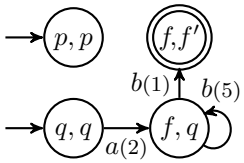
(b) NFA $\mathcal{A}'$ from Figure 4b



(c) Weights (in brackets) on NFA $\mathcal{A}'_{pre^*}$



(d) NFA $\mathcal{A}_2$ for $\{\langle q, ab^k\rangle \mid k \geq 0\}$



(e) Intersection of $\mathcal{A}_2$ and $\mathcal{A}'_{pre^*}$

$$(q,q) \xrightarrow{a(2)} (f,q) \xrightarrow{b(1)} (f,f')$$
$$\langle q, ab\rangle \overset{2}{\Rightarrow} \langle q, b\rangle \overset{1}{\Rightarrow} \langle p, a\rangle$$

(f) Shortest path (top) in product automaton (e) and corresponding trace (bottom) in the WPDA (a)

$$(q,q) \xrightarrow{a(2)} (f,q) \xrightarrow{b(5)} \dots \xrightarrow{b(5)} (f,q) \xrightarrow{b(1)} (f,f')$$
$$\langle q, abb^k\rangle \overset{2}{\Rightarrow} \Big[\langle q, bb^k\rangle \overset{1}{\Rightarrow} \langle p, ab^k\rangle \overset{0}{\Rightarrow} \langle q, aab^k\rangle \overset{2}{\Rightarrow} \langle q, ab^k\rangle \overset{2}{\Rightarrow} \langle q, b^k\rangle\Big] \dots \langle q, b\rangle \overset{1}{\Rightarrow} \langle p, a\rangle$$

(g) The product automaton (e) has a positive cycle, so the longest path (top) is arbitrarily long (repetition is shown by '. . . '). Corresponding longest trace (bottom) in the WPDA (a) is also unbounded.

Fig. 8: Shortest (f) and longest (g) trace analysis from $\mathcal{A}_2$ (d) to $\mathcal{A}'$ (b) through a WPDA (a). In this particular case, $\mathcal{A}'_{pre^*}$ (c) is assigned the same weights for both shortest and longest trace analysis.

configuration in the saturated automaton equals the weight of a shortest trace in the WPDA between that configuration and a configuration in the original NFA.

Figure 8 presents the execution of a shortest and longest trace analysis for the running example, where we assign integer weights (0, 1, and 2) to the rules of the WPDA in Figure 8a. Again we apply pre^* to the NFA $\mathcal{A}'$ shown in Figure 8b that accepts only $\langle p, a\rangle$, but this time the saturation procedure also computes weights for the transitions in $\mathcal{A}'_{pre^*}$ as shown in Figure 8c. Interestingly, in this specific simple example the weights on the transitions in $\mathcal{A}'_{pre^*}$ happen to be the same for both the shortest trace and longest trace analysis. This is not the case in general, where longest trace analysis even needs a different algorithm than when computing the shortest trace (with nonnegative weights).

For the shortest trace analysis with totally ordered, nonnegative weights, we use a priority queue to determine the order in which transitions are added to the automata in the saturation algorithms, similar to how one may implement Dijkstra's famous algorithm [38, 42] for finding the shortest path in a finite graph. Schwoon [148] describes this algorithm for $post^*$, and later work [138] presents both pre^* and $post^*$ algorithms for the more general bounded idem-

potent semirings[2], where the generality comes with the cost of less efficient algorithms that must be used in place of the priority queue.

In Paper E, we implement not only pre^* and $post^*$ with the efficient priority queue algorithms, we also generalize $dual^*$ to work for weighted pushdown automata. In this setting, the on-the-fly product construction cannot return immediately when an accepting path is found. It must instead keep track of the weight of the shortest accepting path and compare this to the smallest weight in the current priority queues used by pre^* and $post^*$.

To complete the running example, we use a set of initial configurations where the control location is q and each stack matches the regular expression ab^* as modelled by the NFA $\mathcal{A}_2$ in Figure 8d. The product automaton shown in Figure 8e encodes the intersection of $\mathcal{A}_2$ and $\mathcal{A}'_{pre^*}$ and assigns weights to transitions as the sum of the corresponding transitions in $\mathcal{A}_2$ and $\mathcal{A}'_{pre^*}$. From the shortest path in the product automaton, we reconstruct in Figure 8f the shortest trace from $\mathcal{A}_2$ to $\mathcal{A}'$, which has the weight 3. In this example the longest trace is unbounded, i.e. for any length we can find a longer trace. This is due to a positive cycle as shown in Figure 8g.

To find the shortest trace when allowing negative weights (and symmetrically to find the longest trace), the problem of infinite traces arise due to the possibility of negative (and symmetrically positive) cycles. For weighted finite graphs, this issue is handled in the well known Bellman-Ford algorithm [38] by relaxing edges, i.e. changing edge weights, a certain number of times after which any further change is due to a negative cycle. A similar idea is presented by Kühnrich et al. [100] for WPDA with unbounded weights as a generalization of the work on bounded weight domains [138].

In PDAAAL, we implement both pre^* and $post^*$ for unbounded, totally ordered weight domains by using a queue with a special token to count the number of weight relaxations and eventually detect a possible infinite trace. Unfortunately, these algorithms cannot easily benefit from the early termination and bidirectional search idea of $dual^*$. Instead we implement a version that interleaves the execution of the pre^* and $post^*$ algorithms to still benefit from the observation that often one of the two search spaces are significantly smaller than the other.

As the example in Figure 8 shows, it can happen that the saturation algorithm terminates without a finding cycle in the pushdown rules, but the product automaton has a cycle that leads to an arbitrarily long trace. Such a cycle in the finite automaton is detected by the Bellman-Ford algorithm [38].

To test the performance of PDAAAL, we benchmark against WALi [93], the state-of-the-art tool for WPDA analysis, on both shortest and longest latency analysis on a large set of MPLS verification queries. We use AalWiNes as the

[2]Bounded idempotent semirings are not necessarily totally ordered, but repeated addition must eventually reach a fixed point.

verification frontend and use respectively PDAAAL or WALi for the WPDA verification backend. Paper E reports the results of the benchmarking and shows that for both shortest and longest trace verification PDAAAL's $dual^*$ algorithm performs best and around two orders of magnitude faster than WALi.

> **Contribution 8 (Paper E)**
> We present PDAAAL, an efficient C++ library and command-line tool for reachability analysis of weighted pushdown automata including shortest and longest trace analysis. We generalize the novel dual* technique to the weighted setting and demonstrate through experimental evaluation a significant runtime improvement of two orders of magnitude compared to the state-of-the-art.

7.4 Formal Correctness of Pushdown Verification

In the preceding sections we have developed a verification approach for MPLS networks through the formal model of pushdown automata, we have designed efficient translations and reachability algorithms, and we have constructed the model checking tool PDAAAL and built AalWiNes on top of it. While formal verification and model checking promises safe systems by exhaustively checking specifications in all scenarios, the guarantees given by model checking are only as strong as the model checking tool used; and like all other software, these tools can be error-prone.

To increase the trust of PDAAAL, we formally verify, in Paper F, the correctness of the pushdown reachability algorithms using the proof assistant Isabelle/HOL [124]. Proof assistants like Isabelle/HOL are typically built around a small trusted proof-checking kernel and a few well-established mathematical axioms from which rich mathematical theories are built and the proofs of all lemmas and theorems rigorously verified by the proof-checker. This structure of machine checkable proofs and a small trusted computing base provides a high level of trust for the results of these proof assistants. Isabelle/HOL is an interactive theorem prover—its expressive higher order logic (HOL) often requires human interaction in constructing the proofs as opposed to the push-button approaches of model checkers, such as PDAAAL, and automated theorem provers like e.g. SMT solvers.

Using Isabelle/HOL, we formalize pushdown automata, finite automata, and the pushdown reachability problem. Next, we formalize and prove the correctness of the pre^* and $post^*$ saturation procedures. We also formalize finite automata intersection and prove the correctness property of $dual^*$. Paper F describes the formalization effort involving around 4.400 lines of Isabelle definitions and proofs, of which a small snippet is shown in Figure 9.

inductive pre_star_rule **where**
$(\mathsf{Init}\ p, \gamma, q) \notin A \longrightarrow (p,\ \gamma) \hookrightarrow (p',\ w)\ \longrightarrow$
$(\mathsf{Init}\ p', \mathsf{lbl}\ w, q) \in \mathsf{LTS.trans\_star}\ A \longrightarrow$
$\mathsf{pre\_star\_rule}\ A\ (A \cup \{(\mathsf{Init}\ p, \gamma, q)\})$

theorem *pre_star_rules_correct*:
assumes $\mathsf{inits} \subseteq \mathsf{LTS.srcs}\ A$ **and** $\mathsf{saturation}\ \mathsf{pre\_star\_rule}\ A\ A'$
shows $\mathsf{lang}\ A' = \mathsf{pre\_star}\ (\mathsf{lang}\ A)$

theorem *pre_star_exec_language_correct*:
assumes $\mathsf{inits} \subseteq \mathsf{LTS.srcs}\ A$
shows $\mathsf{lang}\ (\mathsf{pre\_star\_exec}\ A) = \mathsf{pre\_star}\ (\mathsf{lang}\ A)$

Fig. 9: Isabelle/HOL definition of pre^* saturation rule and correctness theorems from Paper F

These formal proofs give us trust that, at a high level, the algorithms used by PDAAAL for pushdown reachability are correct; however, it does not guarantee the correctness of the low-level implementation. Efficient model checkers, like PDAAAL, use various implementation tricks and specialized data structures to optimize the performance. While it is possible to formally verify efficient implementations, as witnessed by e.g. the verified C complier CompCert [106] and the verified OS microkernel seL4 [94], these projects require a large amount of human labor to develop the formal proofs.

Instead, we take a different approach by using the functional programming language included in Isabelle/HOL to create a formally verified implementation of the pre^* algorithm for pushdown reachability with comparatively little effort. This is the functional program pre_star_exec in Figure 9. We can then extract this as an executable program using Isabelle/HOL's code generation feature [70]. This correct-by-construction program runs slowly compared to the efficient C++ implementation in PDAAAL due to less efficient data structures and e.g. the lack of early termination, but it gives us a formally verified oracle to compare the results of PDAAAL against.

Contribution 9 (Paper F)
We present a formalization in Isabelle/HOL of pushdown reachability algorithms along with machine checkable correctness proofs of the algorithms including the novel bidirectional search algorithm. From this, we generate correct-by-construction code to serve as a formally verified oracle for pushdown reachability.

To increase the trust in the results produced by PDAAAL's efficient C++ implementation and possibly find bugs, we execute the same pushdown reachability problems on PDAAAL and the slow but verified Isabelle/HOL

implementation and compare the results—a technique known as differential testing [49, 69, 116]. Contrary to the standard differential testing of two different untrusted implementations, we here compare against a formally verified implementation, which means we can have a high degree of trust in the correctness of matching results.

Given that there are infinitely many possible pushdown reachability problems, differential testing cannot provide complete guarantees of the correctness of PDAAAL; but with an exhaustive enumeration of (close to 27 million) small cases and additional tests on a large number of both random test cases and network verification problems, we gain a good coverage of both corner cases and real-world cases, and we significantly increase the reliability of PDAAAL.

While PDAAAL showed no discrepancies with the verified oracle on the network verification problems, the differential testing on the random instances and the exhaustively enumerated small instances revealed in total three non-trivial implementation bugs in PDAAAL as reported in Paper F.

From the test cases that show discrepancies in differential testing, we need to locate and fix the implementation error. To aid in this process, we use delta-debugging [177] to minimize the test case by removing parts of the test case, such as pushdown rules, in a structured way and iteratively checking if the smaller test case still produces a violation. As shown in Paper F, this automatic technique significantly simplifies the error-revealing test case making it easier to locate the bug. After fixing the three aforementioned bugs in PDAAAL, the differential testing finds no more discrepancies with the results of the formally verified oracle.

> **Contribution 10 (Paper F)**
> We develop a methodology for increasing the trust in pushdown automata model checking tools by means of differential testing against a formally verified oracle and counterexample minimization. We demonstrate the utility of the approach by discovering and fixing tricky bugs in the implementation of the model checker PDAAAL.

8 Measuring Real Networks

In all the experiments with network resilience and verification so far, we have been using synthetic data planes generated using standard protocols and a dataset of real network topologies. While this can give reasonable indications of the possible real-world performance of the developed tools and techniques, for a real deployment of network verification and synthesis, we need to obtain the actual parameters of the model from the running network. For quantitative

reasoning about e.g. the risk of congestion, we need to know values like link capacities, load balancing ratios, traffic demand and link utilization, as well as how they change over time.

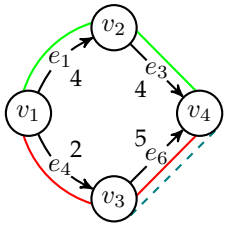
In Paper G, we investigate this in collaboration with a network analytics company, Benocs, and their traffic analytics deployment on a major European ISP network. It turns out that in their analytics there is a discrepancy between the predicted amount of traffic from each flow on a link and the actual measured total traffic on the link—indicating an error in the measurements or analysis. This type of erroneous data could cause quantitative network verification and synthesis tools to provide incorrect or suboptimal answers, and improving the accuracy of scalable traffic analysis is hence a key part of enabling quantitative network verification and synthesis.

The error in this case is caused by incomplete knowledge of the flow splitting ratios that the network uses to load balance traffic among multiple paths, which in turn affects the traffic flow on each link. In a large and heterogeneous network, it is often too difficult to directly access all the technical, vendor-specific implementation details and router configurations needed to determine these flow splitting ratios—obtaining this fine-grained information across the whole network would require a very complex system, and the network analytics company deem it impossible in practice. Instead we propose in Paper G a technique to estimate the flow splitting ratios from the information available to the analytics system, i.e. the utilization of links, the demand of traffic flows from source to destination, and the set of paths used by each flow.

Figure 10 shows a small example of this information. In the network topology in Figure 10a, links are annotated with their current link utilization, and Figure 10b lists two flows with their respective demands and set of paths. Note how the flow f_1 splits its traffic among two paths. The problem is now to determine the ratios by which this traffic is split based on the available information, i.e. the link utilization, flow demands, and paths.

The solution we propose is to formulate the problem as a quadratic linear program that finds flow splitting ratios that minimizes the error of estimation. Figure 10c shows the quadratic program for our small example, where the optimal solution is that flow f_1 sends $x_{e_1}^{f_1} = 2/3$ of the traffic on link e_1 and $x_{e_4}^{f_1} = 1/3$ on link e_4. This simple example is particularly nice, as there exists an exact solution with no error. In the real world, this is seldom the case.

In the dataset from the large ISP network, we observe that small-scale noise is common due to e.g. timing and sampling variance, but we also observe cases of large (but relatively rare) errors in the flow measurement due to e.g. late detection of changes in the BGP routing tables. In Paper G, we handle these errors by filtering out the link constraints in the quadratic program with the highest errors—assuming that these correspond to erroneous measurements—as well as combining data from multiple time slots with the assumption that the flow splitting ratios remain stable over time.



(a) Part of the network topology from Figure 1a with link utilizations and flows

Flow	Demand	Paths
$f_1 : v_1 \to v_4$	6	$v_1 - v_2 - v_4$ $v_1 - v_3 - v_4$
$f_2 : v_3 \to v_4$	3	$v_3 - v_4$

(b) Two flows through the network and their traffic demands and paths. The first flow splits its traffic among two different paths.

Define non-negative variables:

$$x_{e_1}^{f_1}, x_{e_3}^{f_1}, x_{e_4}^{f_1}, x_{e_6}^{f_1}, x_{e_6}^{f_2}, err_{e_1}, err_{e_3}, err_{e_4}, err_{e_6}$$

Minimize:

$$(err_{e_1})^2 + (err_{e_3})^2 + (err_{e_4})^2 + (err_{e_6})^2$$

Subject to:

$$x_{e_1}^{f_1} + x_{e_4}^{f_1} = 1 \qquad x_{e_1}^{f_1} = x_{e_3}^{f_1} \qquad x_{e_4}^{f_1} = x_{e_6}^{f_1} \qquad x_{e_6}^{f_2} = 1$$

$$err_{e_1} \cdot 4 \geq 6 \cdot x_{e_1}^{f_1} - 4 \qquad err_{e_3} \cdot 4 \geq 6 \cdot x_{e_3}^{f_1} - 4$$

$$err_{e_4} \cdot 2 \geq 6 \cdot x_{e_4}^{f_1} - 2 \qquad err_{e_6} \cdot 5 \geq 6 \cdot x_{e_6}^{f_1} + 3 \cdot x_{e_6}^{f_2} - 5$$

(c) Quadratic programing formulation for finding flow splitting ratios given link utilizations and flow demands

Fig. 10: Example estimation of flow splitting ratios given link utilizations, flow demands, and paths

We demonstrate the accuracy and robustness of our method through synthetic experiments, where we can control the flow splitting ratios, and we show its scalability by application to real traffic data from a large ISP network with over 3 000 routers and 14 000 links, where flows carrying 99.9% of the total traffic volume is analyzed in 87 seconds on a standard laptop. The later experiment shows that flow splitting ratios based on link capacities give more accurate predictions for the ISP network—an insight now used in production by the network analytics company.

Finally, the dataset can possibly be used for future experiments with the performance of quantitative verification and synthesis for networks.

Contribution 11 (Paper G)
We present a technique for estimating the flow splitting ratios that a network uses to load balance traffic. We show how to handle noisy, incomplete or erroneous measurements of link utilization, traffic demands, and flow paths. We demonstrate the accuracy through simulation, and we show the technique's applicability on real data from a large ISP network.

9 Conclusion

With the work presented in this thesis, we take a step towards the correct automated operation of efficient and resilient computer networks.

We formalize the widely-deployed MPLS networks with a focus on resilience and multipath routing, and we use the formal model in the design of R-MPLS—a technique for synthesizing loop-free recursive fast reroute protection for MPLS networks, which provides increased resilience to multiple failures with only a modest memory and communication overhead.

Prior work [80] implements qualitative verification of MPLS networks using the connection between MPLS networks and pushdown automata [144] and a query language based on regular expressions. We extend this approach to quantitative MPLS network verification with shortest and longest trace analysis for a variety of possible weights. Moreover, we significantly improve the runtime efficiency through the development of new on-the-fly algorithms and efficient C++ implementations in the tools PDAAAL and AalWiNes. This efficient verification is needed to make the tools usable in practice.

We increase the trust in our pushdown automata model checking tool PDAAAL by formalizing and rigorously proving in Isabelle/HOL the correctness of the used algorithms, extracting a verified implementation, and performing differential testing against it. The methodology includes differential testing against a formally verified oracle and subsequent counterexample minimization to ease the bug finding. We successfully apply this to detect and fix three non-trivial errors in PDAAAL.

Given the lack of publicly available datasets of MPLS network data planes, we enable realistic experimental evaluation of our ideas through the development of the tool MPLS-Kit that can synthesize and simulate MPLS network data planes using industry-standard protocols.

For practical use of network verification and synthesis, the formal network models need to be populated with data from the running network; and particularly for quantitative reasoning, accurate and detailed traffic data is needed. Traffic measurement solutions need to be scalable and are hence based on sampling at the flow level. We show how to derive flow splitting ratios, which are otherwise unknown, from link utilizations and the demand and paths of each traffic flow—even handling the noisy and erroneous measurements found in a real traffic analytics deployment on a large European ISP network.

Future work: There are several directions of future work from this thesis. For the quantitative synthesis of protection schemes, it is interesting to extend the R-MPLS technique with multipath protection and optimize resilience and link utilization based on expected traffic patterns to get congestion-aware, loop-free, recursive protection. Another extension of R-MPLS is to apply the technique to segment routing, where the source routing scheme gives different

memory trade-offs to consider. Segment routing is also an interesting target for the pushdown-based network verification in AalWiNes.

Currently AalWiNes supports existential queries: "does there exist a failure scenario, where there is a trace matching a regular pattern?" An interesting extension of this asks if for all considered failure scenarios there exist a trace matching the regular pattern. The quantifier alternation of such queries hints at the use of alternating pushdown automata, for which reachability algorithms are known [22]. To also perform quantitative analysis of such queries, investigation into a weighted extension of alternating pushdown automata is interesting. A natural next step for the Isabelle/HOL formalization of pushdown algorithms is to prove the correctness of the weighted pre^*, $post^*$, and $dual^*$, extract a formally verified implementation, and perform differential testing on the weighted algorithms in PDAAAL.

To improve the efficiency and robustness of the simulator in MPLS-Kit, future work could investigate WPDA algorithms for efficiently computing the link utilizations of flows in an MPLS network with the behavior of stack operations and possibility of loops. This may require the development of WPDA algorithms that work on domains where the operation that combines weights of multiple paths is not idempotent, such as e.g. integer addition, which current algorithms are not designed for.

References

[1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "AED: Incrementally synthesizing policy-compliant and manageable configurations," in *Proc. ACM CoNEXT*, 2020, pp. 482–95.

[2] ——, "Tiramisu: Fast multilayer network verification," in *Proc. USENIX NSDI*, 2020, pp. 201–219.

[3] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proc. ACM Workshop on Assurable and Usable Security Configuration (SafeConfig)*, 2010, pp. 37–44.

[4] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: towards end-to-end verification of network reachability and security," in *IEEE International Conference on Network Protocols*, 2009, pp. 123–132.

[5] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, "Modular control plane verification via temporal invariants," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, article 108, Jun. 2023, 26 pages.

[6] A. Alshnakat, D. Lundberg, R. Guanciale, M. Dam, and K. Palmskog, "HOL4P4: Semantics for a verified data plane," in *Proc. 5th International Workshop on P4 in Europe (EuroP4 '22)*. ACM, 2022, pp. 39–45.

[7] Amazon Web Services, "Summary of the Amazon S3 service disruption in the northern virginia (US-EAST-1) region," https://aws.amazon.com/message/41926/, 2017.

[8] ——, "Summary of the AWS service event in the northern virginia (US-EAST-1) region," https://aws.amazon.com/message/12721/, Dec. 2021.

[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *Proc. ACM POPL*, 2014, pp. 113–126.

[10] L. Andersson, I. Minei, and B. Thomas, "LDP specification," RFC 5036, Oct. 2007.

[11] M. T. Arashloo, R. Beckett, and R. Agarwal, "Formal methods for network performance analysis," in *Proc. USENIX NSDI*, 2023, pp. 645–661.

[12] D. O. Awduche, L. Berger, D.-H. Gan, T. Li, D. V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP tunnels," RFC 3209, Dec. 2001.

[13] BBC News, "Human error behind network outage, Sure confirms," Feb. 2023. [Online]. Available: https://www.bbc.com/news/world-europe-guernsey-64511077

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. ACM SIGCOMM*, 2017, pp. 155–168.

[15] ——, "Control plane compression," in *Proc. ACM SIGCOMM*, 2018, pp. 476–489.

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

[17] ——, "Network configuration synthesis with abstract topologies," in *Proc. ACM PLDI*, 2017, pp. 437–451.

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

[19] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Config2Spec: Mining network specifications from network configurations," in *Proc. USENIX NSDI*, 2020, pp. 969–984.

[20] J. Bogle, N. Bhatia, M. Ghobadi, I. Menache, N. Bjørner, A. Valadarsky, and M. Schapira, "TEAVAR: Striking the right utilization-availability balance in WAN traffic engineering," in *Proc. ACM SIGCOMM*, 2019, pp. 29–43.

[21] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[22] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[23] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *IEEE Local Computer Network Conference*, 2010, pp. 408–415.

[24] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2016.

[25] B. Brown, "Level 3 blames huge network outage on human error," *Network World*, Oct. 2016. [Online]. Available: https://www.networkworld.com/article/3128104/

[26] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[27] J. R. Büchi, "Regular canonical systems," *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[28] R. Callon, "Use of OSI IS-IS for routing in TCP/IP and dual environments," RFC 1195, Dec. 1990.

[29] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A protection architecture for enterprise networks," in *Proc. 15th Conf. USENIX Security Symp.*, vol. 15, 2006, pp. 137–151.

[30] F. Chang, K. Onohara, and T. Mizuochi, "Forward error correction for 100 G transport networks," *IEEE Communications Magazine*, vol. 48, no. 3, pp. S48–S55, 2010.

[31] Y. Chang, C. Jiang, A. Chandra, S. Rao, and M. Tawarmalani, "Lancet: Better network resilience by designing for pruned failure sets," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, article 49, Dec. 2019, 26 pages.

[32] M. Chiesa, A. Kamisinski, J. Rak, G. Retvari, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys and Tutorials (COMST)*, pp. 1253–1301, 2021.

[33] N. Christensen, M. Glavind, S. Schmid, and J. Srba, "Latte: Improving the latency of transiently consistent network update schedules," *SIGMETRICS Perform. Eval. Rev.*, vol. 48, no. 3, pp. 14–26, Mar. 2021.

[34] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 1855. Springer, 2000, pp. 154–169.

[35] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.

[36] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[37] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 4th ed. MIT press, 2022.

[39] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a network service," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1726–1738, 2020.

[40] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, ser. LNCS, vol. 4963. Springer Berlin Heidelberg, 2008, pp. 337–340.

[41] M. Didriksen, P. G. Jensen, J. F. Jønler, A.-I. Katona, S. D. L. Lama, F. B. Lottrup, S. Shajarat, and J. Srba, "Automatic synthesis of transiently correct network updates via Petri games," in *Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021)*, ser. LNCS, vol. 12734. Springer, 2021, pp. 118–137.

[42] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[43] R. Doenges, M. T. Arashloo, S. Bautista, A. Chang, N. Ni, S. Parkinson, R. Peterson, A. Solko-Breslin, A. Xu, and N. Foster, "Petr4: Formal foundations for P4 data planes," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, article 41, Jan. 2021, 32 pages.

[44] R. Doenges, T. Kappé, J. Sarracino, N. Foster, and G. Morrisett, "Leapfrog: Certified equivalence for protocol parsers," in *Proc. ACM PLDI*, 2022, pp. 950–965.

[45] I. v. Duijn, P. G. Jensen, J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "Automata-theoretic approach to verification of MPLS networks under link failures," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 766–781, 2022.

[46] M. Eichholz, E. H. Campbell, M. Krebs, N. Foster, and M. Mezini, "Dependently-typed data plane programming," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, article 40, Jan. 2022, 28 pages.

[47] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *CAV'17*, ser. LNCS, vol. 10427. Springer, 2017, pp. 261–281.

[48] ——, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. USENIX NSDI*, 2018, pp. 579–594.

[49] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *ESEC-FSE 2007*. ACM, 2007, pp. 549–552.

[50] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. USENIX OSDI*, 2016, pp. 217–232.

[51] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *Proc. USENIX NSDI*, 2016, pp. 275–289.

[52] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis, "The lockdown effect: Implications of the COVID-19 pandemic on internet traffic," in *Proc. ACM Internet Measurement Conference (IMC)*, 2020, pp. 1–18.

[53] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[54] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment routing architecture," RFC 8402, Jul. 2018.

[55] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," in *INFINITY'97*, ser. ENTCS, vol. 9. Elsevier, 1997, pp. 27–37.

[56] K. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1435–1461, 2019.

[57] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. USENIX NSDI*, 2015, pp. 469–483.

[58] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional IP routing protocols," *IEEE Communications Magazine*, vol. 40, no. 10, pp. 118–124, 2002.

[59] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison, "Languages for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 128–134, 2013.

[60] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. ACM International Conference on Functional Programming (ICFP)*, 2011, pp. 279–291.

[61] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic NetKAT," in *Programming Languages and Systems (ESOP 2016)*, ser. LNCS, vol. 9632. Springer Berlin Heidelberg, 2016, pp. 282–309.

[62] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for NetKAT," in *Proc. ACM POPL*, 2015, pp. 343–355.

[63] V. Fuller and T. Li, "Classless inter-domain routing (CIDR): The internet address assignment and aggregation plan," RFC 4632, Aug. 2006.

[64] T. Gehr, S. Misailovic, P. Tsankov, L. Vanbever, P. Wiesmann, and M. Vechev, "Bayonet: Probabilistic inference for networks," in *Proc. ACM PLDI*, 2018, pp. 586–602.

[65] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM*, 2016, pp. 300–313.

[66] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of TCP," in *Proc. Symposium on SDN Research (SOSR '17)*. ACM, 2017, pp. 61–74.

[67] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "NV: An intermediate language for verification of network control planes," in *Proc. ACM PLDI*, 2020, pp. 958–973.

[68] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz, "Quality of service terminology in IP networks," *IEEE Communications Magazine*, vol. 41, no. 3, pp. 153–159, 2003.

[69] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *ICSE 2007*. IEEE Computer Society, 2007, pp. 621–631.

[70] F. Haftmann and T. Nipkow, "Code generation via higher-order rewrite systems," in *FLOPS 2010*, ser. LNCS, M. Blume, N. Kobayashi, and G. Vidal, Eds., vol. 6009. Springer, 2010, pp. 103–117.

[71] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[72] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[73] C. Hopps, "Analysis of an equal-cost multi-path algorithm," RFC 2992, Nov. 2000.

[74] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *Proc. USENIX NSDI*, 2017, pp. 735–749.

[75] International Telecommunication Union, "Forward error correction for submarine systems," *ITU-T Recommendation*, vol. G.975, 2000.

[76] ISO, "Intermediate system to intermediate system intra-domain routeing exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)," ISO/IEC 10589:2002, Nov. 2002.

[77] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.

[78] S. Janardhan, "Update about the october 4th outage," *Engineering at Meta*, Oct. 2021. [Online]. Available: https://engineering.fb.com/2021/10/04/networking-traffic/outage/

[79] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma, "Validating datacenters at scale," in *Proc. ACM SIGCOMM*, 2019, pp. 200–213.

[80] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[81] P. G. Jensen, S. Schmid, M. K. Schou, J. Srba, J. Vanerio, and I. van Duijn, "Faster pushdown reachability analysis with applications in network verification," in *Automated Technology for Verification and Analysis (ATVA 2021)*, ser. LNCS, vol. 12971. Springer, 2021, pp. 170–186.

[82] P. G. Jensen, S. Schmid, M. K. Schou, and J. Srba, "PDAAAL: A library for reachability analysis of weighted pushdown systems," in *Automated Technology for Verification and Analysis (ATVA 2022)*. Springer, 2022, pp. 225–230.

[83] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[84] C. Jiang, Z. Li, S. Rao, and M. Tawarmalani, "Flexile: Meeting bandwidth objectives almost always," in *Proc. ACM CoNEXT*, 2022, pp. 110–125.

[85] C. Jiang, S. Rao, and M. Tawarmalani, "PCF: Provably resilient flexible routing," in *Proc. ACM SIGCOMM*, 2020, pp. 139–153.

[86] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, 2014, pp. 539–550.

[87] N. S. Johansen, L. B. Kær, A. L. Madsen, K. O. Nielsen, J. Srba, and R. G. Tollund, "Kaki: Efficient concurrent update synthesis for SDN," *Form. Asp. Comput.*, Jun. 2023.

[88] Juniper Networks, "IS-IS user guide: Understanding weighted ECMP traffic distribution on one-hop IS-IS neighbors," Jan. 2021. [Online]. Available: https://www.juniper.net/documentation/us/en/software/junos/is-is/topics/concept/wecmp-for-one-hop-isis-neighbors-overview.html

[89] D. Katz and D. Ward, "Bidirectional forwarding detection (BFD)," RFC 5880, Jun. 2010.

[90] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. USENIX NSDI*, 2013, pp. 99–111.

[91] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012, pp. 113–126.

[92] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. USENIX NSDI*, 2013, pp. 15–27.

[93] N. Kidd, A. Lal, and T. Reps, "WALi: The weighted automaton library," 2007. [Online]. Available: https://research.cs.wisc.edu/wpis/wpds/wali/

[94] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal verification of an OS kernel," in *Proc. ACM SOSP*, 2009, pp. 207–220.

[95] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[96] D. Kozen, "Kleene algebra with tests," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 427–443, May 1997.

[97] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[98] U. Krishnaswamy, R. Singh, P. Mattes, P.-A. C. Bissonnette, N. Bjørner, Z. Nasrin, S. Kothari, P. Reddy, J. Abeln, S. Kandula, H. Raj, L. Irun-Briz, J. Gaudette, and E. Lan, "OneWAN is better than two: Unifying a split WAN architecture," in *Proc. USENIX NSDI*, 2023, pp. 515–529.

[99] D. Kristiansen and M. K. Schou, "Quantitative analysis of MPLS networks in AalWiNes: Shortest trace reachability analysis of weighted pushdown systems," Master's thesis, Aalborg University, 2020.

[100] M. Kühnrich, S. Schwoon, J. Srba, and S. Kiefer, "Interprocedural dataflow analysis over weight domains with infinite descending chains," in *FOSSACS'09*, ser. LNCS, vol. 5504. Springer-Verlag, 2009, pp. 440–455.

[101] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé, "Semi-oblivious traffic engineering: The road not taken," in *Proc. USENIX NSDI*, 2018, pp. 157–170.

[102] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliability Engineering & System Safety*, vol. 105, pp. 104–113, 2012.

[103] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba, "AllSynth: A BDD-based approach for network update synthesis," *Science of Computer Programming*, vol. 230, article 102992, 2023, 19 pages.

[104] K. G. Larsen, S. Schmid, and B. Xue, "WNetKAT: A weighted SDN programming and verification language," in *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, ser. LIPIcs, vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 18:1–18:18.

[105] S. Lee, K. Levanti, and H. S. Kim, "Network monitoring: Present and future," *Computer Networks*, vol. 65, pp. 84–98, 2014.

[106] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[107] J. Li, "Rogers blames massive outage on error during network update," *CBC News*, Jul. 2022. [Online]. Available: https://www.cbc.ca/news/business/rogers-letter-outage-crtc-1.6530067

[108] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proc. ACM SIGCOMM*, 2014, pp. 527–538.

[109] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "ZUpdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, 2013, pp. 411–422.

[110] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully emulating large production networks," in *Proc. ACM SOSP*, 2017, pp. 599–613.

[111] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: Practical verification for programmable data planes," in *Proc. ACM SIGCOMM*, 2018, pp. 490–503.

[112] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. USENIX NSDI*, 2015, pp. 499–512.

[113] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013, article 20, pp. 1–7.

[114] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, 2011, pp. 290–301.

[115] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *Proc. ACM PLDI*, 2015, pp. 196–207.

[116] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[117] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[118] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. ACM POPL*, 2012, pp. 217–230.

[119] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–13.

[120] J. Moy, "OSPF version 2," RFC 2328, Apr. 1998.

[121] K. S. Namjoshi, "Certifying model checkers," in *Computer Aided Verification*. Springer, 2001, pp. 2–13.

[122] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008.

[123] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of P4 programs in feasible time using assertions," in *Proc. ACM CoNEXT*, 2018, pp. 73–85.

[124] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[125] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," RFC 4090, May 2005.

[126] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proc. USENIX NSDI*, 2017, pp. 699–718.

[127] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai, "A survey on low latency towards 5G: RAN, core network and caching solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3098–3130, 2018.

[128] R. Peterson, E. H. Campbell, J. Chen, N. Isak, C. Shyu, R. Doenges, P. Ataei, and N. Foster, "P4Cub: A little language for big routers," in *Conference on Certified Programs and Proofs (CPP 2023)*. ACM, 2023, pp. 303–319.

[129] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. ACM POPL*, 2016, pp. 69–83.

[130] J. Postel, "Internet protocol," RFC 791, Sep. 1981.

[131] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *Proc. USENIX NSDI*, 2020, pp. 953–967.

[132] Public Safety and Homeland Security Bureau, "FCC report on nationwide CenturyLink network outage on december 27, 2018," *Federal Communications Commission*, Aug. 2019. [Online]. Available: https://www.fcc.gov/document/fcc-report-centurylink-network-outage

[133] A. Raj and O. C. Ibe, "A survey of IP and multiprotocol label switching fast reroute schemes," *Computer Networks*, vol. 51, no. 8, pp. 1882–1907, 2007.

[134] J. Rak and D. Hutchison, *Guide to disaster-resilient communication networks*. Springer Nature, 2020.

[135] S. Ramanathan, Y. Zhang, M. Gawish, Y. Mundada, Z. Wang, S. Yun, E. Lippert, W. Taha, M. Yu, and J. Mirkovic, "Practical intent-driven routing configuration synthesis," in *Proc. USENIX NSDI*, 2023, pp. 629–644.

[136] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 323–334, Aug. 2012.

[137] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," RFC 4271, Jan. 2006.

[138] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Science of Computer Programming*, vol. 58, no. 1-2, pp. 206–263, 2005.

[139] N. Rockwell, "Summary of june 8 outage," *Fastly*, Jun. 2021. [Online]. Available: https://www.fastly.com/blog/summary-of-june-8-outage

[140] E. C. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," RFC 3031, Jan. 2001.

[141] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, "Correct by construction networks using stepwise refinement," in *Proc. USENIX NSDI*, 2017, pp. 683–698.

[142] S. Saha, S. Prabhu, and P. Madhusudan, "NetGen: Synthesizing data-plane configurations for network policies," in *Proc. ACM SOSR*, 2015, article 17, pp. 1–6.

[143] A. Schlichtkrull, M. K. Schou, J. Srba, and D. Traytel, "Differential testing of pushdown reachability with a formally verified oracle," in *Proc. Formal Methods in Computer-Aided Design (FMCAD)*. TU Wien Academic Press, Oct. 2022, pp. 369–379.

[144] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating MPLS networks," in *Proc. IEEE INFOCOM*, 2018, pp. 1799–1807.

[145] S. Schmid, M. K. Schou, J. Srba, and J. Vanerio, "R-MPLS: Recursive protection for highly dependable MPLS networks," in *Proc. ACM CoNEXT*, 2022, pp. 276–292.

[146] S. Schmid, B. C. Schrenk, and Á. Torralba, "NetStack: A game approach to synthesizing consistent network updates," in *IFIP Networking*, 2022, pp. 1–9.

[147] T. Schneider, R. Birkner, and L. Vanbever, "Snowcap: Synthesizing network-wide configuration updates," in *Proc. ACM SIGCOMM*, 2021, pp. 33–49.

[148] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, 2002.

[149] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[150] A. Sharma, "Microsoft cloud outage hits users around the world," *Reuters*, Jan. 2023. [Online]. Available: https://www.reuters.com/technology/microsoft-teams-down-thousands-users-india-downdetector-2023-01-25/

[151] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. CENGAGE Learning, 2012.

[152] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the forwarding plane of a data-center switch," in *Proc. Symposium on SDN Research (SOSR '15)*. ACM, 2015, article 2, pp. 1–8.

[153] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva, "Cantor meets Scott: Semantic foundations for probabilistic networks," in *Pro. ACM POPL*, 2017, pp. 557–571.

[154] S. Smolka, P. Kumar, D. M. Kahn, N. Foster, J. Hsu, D. Kozen, and A. Silva, "Scalable verification of probabilistic networks," in *Proc. ACM PLDI*, 2019, pp. 190–203.

[155] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *Proc. ACM CoNEXT*, 2014, pp. 213–226.

[156] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proc. ACM SIGCOMM*, 2020, pp. 750–764.

[157] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with Vera," in *Proc. ACM SIGCOMM*, 2018, pp. 518–532.

[158] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM*, 2016, pp. 314–327.

[159] T. Strickx and J. Hartman, "Cloudflare outage on june 21, 2022," *The Cloudflare Blog*, Jun. 2022. [Online]. Available: https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/

[160] K. Subramanian, L. D'Antoni, and A. Akella, "Genesis: Synthesizing forwarding tables in multi-tenant networks," in *Proc. ACM POPL*, 2017, pp. 572–585.

[161] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Kirigami, the verifiable art of network cutting," in *IEEE International Conference on Network Protocols (ICNP)*, 2022, pp. 1–12.

[162] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu, "Aquila: A practically usable verification system for production-scale programmable data planes," in *Proc. ACM SIGCOMM*, 2021, pp. 17–32.

[163] R. J. Vanderbei, *Linear programming: Foundations and Extensions*, 5th ed. Springer, 2020.

[164] J. Vanerio, S. Schmid, M. K. Schou, and J. Srba, "MPLS-Kit: An MPLS data plane toolkit," in *IEEE 11th International Conference on Cloud Networking (CloudNet)*, Nov. 2022, pp. 49–54.

[165] S. Vissicchio and L. Cittadini, "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[166] J. Wagemaker, N. Foster, T. Kappé, D. Kozen, J. Rot, and A. Silva, "Concurrent NetKAT," in *Programming Languages and Systems (ESOP 2022)*, ser. LNCS, vol. 13240. Springer, 2022, pp. 575–602.

[167] Q. Wang, M. Pan, S. Wang, R. Doenges, L. Beringer, and A. W. Appel, "Foundational verification of stateful P4 packet processing," in *Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023, article 32, pp. 32:1–32:20.

[168] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, "R3: Resilient routing reconfiguration," in *Proc. ACM SIGCOMM*, 2010, pp. 291–302.

[169] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Scalable verification of border gateway protocol configurations with an smt solver," in *Proc. ACM OOPSLA*, 2016, pp. 765–780.

[170] G. G. Xie, Jibin Zhan, D. A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *Proc. IEEE INFOCOM*, vol. 3, 2005, pp. 2170–2183.

[171] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2016.

[172] ——, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2900–2915, 2017.

[173] N. Yaseen, B. Arzani, R. Beckett, S. Ciraci, and V. Liu, "Aragog: Scalable runtime verification of shardable networked systems," in *Proc. USENIX OSDI*, 2020, pp. 701–718.

[174] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, D. She, Q. Ma, B. Cheng, H. Xu, M. Zhang, Z. Wang, and R. Fonseca, "Accuracy, scalability, coverage: A practical configuration verifier on a global WAN," in *Proc. ACM SIGCOMM*, 2020, pp. 599–614.

[175] Y. Yuan, D. Lin, R. Alur, and B. T. Loo, "Scenario-based programming for SDN policies," in *Proc. ACM CoNEXT*, 2015, article 34, pp. 1–13.

[176] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A custom symbolic model checker for stateful network verification," in *Proc. USENIX NSDI*, 2020, pp. 181–200.

[177] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[178] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. USENIX NSDI*, 2014, pp. 87–99.

[179] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, "Differential network analysis," in *Proc. USENIX NSDI*, 2022, pp. 601–615.

[180] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *Proc. USENIX NSDI*, 2020, pp. 241–255.

Part II

Papers

Paper A

R-MPLS: Recursive Protection for Highly Dependable MPLS Networks

Stefan Schmid, Morten Konggaard Schou,
Jiří Srba, and Juan Vanerio

The paper has been published in:
Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22), pp. 276-292, Association for Computing Machinery, 2022. https://doi.org/10.1145/3555050.3569140

The layout has been revised.

Abstract

Most modern communication networks feature fast rerouting mechanisms in the data plane. However, design and configuration of such mechanisms even under multiple failures is known to be difficult. In order to increase the resilience of the widely deployed MPLS networks, we propose R-MPLS, an alternative link protection mechanism for MPLS networks that uses recursive *protection and can route around* multiple *simultaneously failed links. Our new R-MPLS approach comes with strong theoretical underpinnings, is implementable in a fully distributed way and executable on existing MPLS hardware, and formally guarantees that no forwarding loops are introduced. We implement our R-MPLS protection in an automated tool which overcomes the complexity of configuring such resilient network data planes, and report on the benefits of recursive protection in realistic network topologies. We find that R-MPLS significantly increases network robustness against multiple failures, with only moderate increase in the number of forwarding rules and communication overhead (both comparable to industry-standards like RSVP-TE FRR).*

1 Introduction

Network failures are inevitable: network interfaces can go down and devices crash at any time [1, 2]. Today, especially link failures are common, and with the increasing scale of communication networks, failures are likely to become more frequent [3]. In order to deal with such failures and provide the required high degree of dependability, modern networks rely on control software whose responsibility is to ensure connectivity despite these unreliable components. In particular, most mission-critical communication networks today feature fast rerouting (FRR) mechanisms in the data plane [4] which allow routers to locally and hence quickly forward traffic to alternative paths.

However, the design of fast rerouting mechanisms providing a high degree of resilience is known to be challenging, and continues to attract significant attention from the research community [4–8]. In particular, since routers need to react to failures *locally*, these decisions are taken without knowledge of potential failures downstream. The additional failures, however, may lead to incorrect forwarding behaviors and threaten reachability.

Many major network outages have been reported over the last years [9, 10]. While sometimes already a single link failure can lead to undesired network behaviors [11], with the increasing network scale and due to shared risk link groups, operators now even have to plan for multiple failures simultaneously [12]. Also, multiple link failures have already been studied in the literature intensively before [8, 13–15]. There results for achieving perfect resilience for many classes of topologies and different kinds of networks. None of them is readily implementable in MPLS.

This paper is motivated by observing an opportunity to significantly improve the resilience provided by fast rerouting mechanisms. In particular, we consider the widely deployed networks based on Multiprotocol Label Switching (MPLS) [16, 17]. For example, MPLS networks are popular among ISPs for traffic engineering purposes. The fast rerouting mechanism used in MPLS relies on stacks of labels in the packet header, where a label pushed on the stack allows to route packets around failed links, creating a "backup tunnel" [18]. MPLS FRR allows protecting against individual link and node failures, and has been successfully used for two decades already. It has recently regained attention for supporting fast what-if analysis [19, 20].

Two common protection methods are standardized on MPLS for protection of traffic engineering tunnels [18]: *one-to-one backup* where one backup label switched path (LSP) is established for each protected LSP in such a way that the former intersects the latter at one of the downstream nodes and *facility backup* where each backup LSP is established to protect a set of many primary LSPs that share the same outgoing interface or next-hop, intersecting the primary paths at a shared downstream node right after the failed link/node. However, while conceptually simple, MPLS FRR procedures are designed to protect against a single link or node failure (as a weakness and in contrast to our method).

Our main contribution is a generalization of the MPLS fast reroute mechanism, R-MPLS, which supports a *recursive* protection scheme, where additional labels are pushed on the stack whenever a packet encounters another failure, essentially creating "nested tunnels". This generalization is non-trivial: if done naively, nested tunnels may quickly lead to forwarding loops, which is a major concern of operators. Also, while R-MPLS may increase the header size, this overhead occurs only when it is needed due to multiple link failures. To this end, we believe that our approach is in line with other trends in networking, such as IPv6 or segment routing [21], which require larger headers. Although MPLS forwarding requires exact matching on labels, which is less expensive than IP ternary matches, the number of routing entries and the number of communications required to compute a protection are critical parameters to scale with the network size. Our recursive protection mechanism R-MPLS then substantially improves the resilience of the network to multiple link failures without the risk of introducing forwarding loops, while keeping low the memory and the communications overheads.

By recursively building protection tunnels, R-MPLS is able to provide alternative paths from a single router, as well as protection paths for other protection paths, neither of which is possible with standard MPLS protection mechanisms. Figure 1 exemplifies the protections provided by R-MPLS. A main path from v_0 to v_1 can be backed up with a protection path via v_2. If (v_0, v_2) is also unavailable then the main path is protected by another protection path via v_3, which rejoins the original protection path at v_2. Additionally

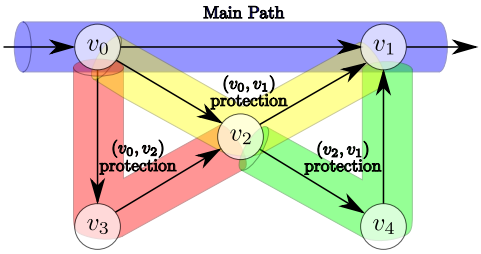


Fig. 1: Example R-MPLS protection.

also link (v_2, v_1) has its own protection via v_4. As a result, R-MPLS can find a path from v_0 to v_1 even when links $(v_0, v_1), (v_0, v_2)$ and (v_2, v_1) simultaneously fail. R-MPLS's improvement is due to recursive link failure protection by design.

We evaluate the benefits of such recursive protection empirically on a large number of real network topologies, also comparing against the state-of-the-art mechanism to achieve multi-failure resiliency. We find that R-MPLS can indeed significantly increase the network resilience against multiple link failures at minimal overheads. Another attractive feature of R-MPLS is that it is compatible with and can be employed on top of any existing MPLS data plane and protocol, such as RSVP and LDP.

As a contribution to the research community, in order to ensure reproducibility and support follow-up work, we make all our experimental artifacts and implementations publicly available (as open-source code) [22].

2 MPLS Network Model

Let us first formally define a general data plane model of MPLS networks. This model is based on prior formal models of MPLS [19, 20], though we restrict the model to the widely used per-platform label space.

In the model, an MPLS network consists of a topology and forwarding rules, where the topology is composed of routers and directed links. Bidirectional links, which are common in real networks, are modelled by two directed links. Figure 2a gives a small example topology. In the figure, links in_1, in_2 and out_1 are connected to the outside of the MPLS domain, modelled using a designated external node not shown in the figure.

Definition 1. A *network topology* is a directed multigraph (V, E, src, tgt) where V is a set of *routers*, E is a set of *links* between routers, $src : E \to V$ assigns the *source router* to each link, and $tgt : E \to V$ assigns the *target router*.

A *path* p in the directed multigraph is a sequence of links $e_1 \dots e_n \in E^*$

with $tgt(e_i) = src(e_{i+1})$ for $1 \leq i < n$. The path is *simple* if all its routers are distinct. Define $tgt(p) \triangleq tgt(e_n)$.

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. In our model, this set does not change for the duration of time considered. In other words we look at a snapshot of the data plane after some failures happen and before the control plane computes new paths. A link is *active* if it belongs to $E \setminus F$. We sometimes call F the failure scenario.

Forwarding in an MPLS network is accomplished using labels in the packet header. We denote the set of MPLS labels used in the network by L. Packet headers are modified using pop, swap and push operations. For a set of MPLS labels L, we define the set of *MPLS operations* on packet headers as $Op(L) = \{\texttt{swap}(\ell) \mid \ell \in L\} \cup \{\texttt{push}(\ell) \mid \ell \in L\} \cup \{\texttt{pop}\}$.

Each router has a mapping from labels to forwarding entries. Figure 2c shows an example of a forwarding table for the topology in Figure 2a. The tables encode two Label Switched Paths (LSPs) from v_1 resp. v_2 and exiting at out_1.

For ease of presentation, the formal model does not include how a packet enters the MPLS domain, but it is easily extendable by also mapping these external interfaces to forwarding entries. In the formal definition, these mappings for all routers are joined into one forwarding table τ:

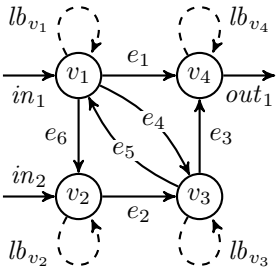
Definition 2. An *MPLS network* $N = (V, E, src, tgt, L, \tau)$ is a tuple where (V, E, src, tgt) is a network topology, L is a finite set of MPLS labels, and $\tau : V \times L \to 2^{\mathbb{N} \times E \times Op(L)^+}$ is the forwarding table.

For every router-label pair $(v, \ell) \in V \times L$, the forwarding table returns a set $\tau(v, \ell) = \{(pr_1, e_1, \omega_1), \ldots, (pr_m, e_m, \omega_m)\}$ of *forwarding entries* where, for all $1 \leq j \leq m$, pr_j is the priority, e_j is the outgoing link such that $src(e_j) = v$, and $\omega_j \in Op(L)^+$ is a nonempty sequence of MPLS operations to be performed on the packet header. We say that a forwarding entry is *active*, if its outgoing link is active.

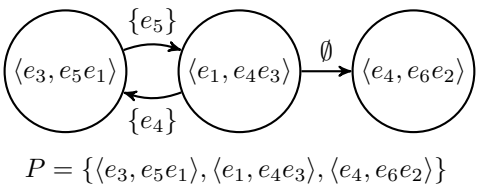
The semantics of a set of forwarding entries is to choose an active entry with the highest priority (lowest natural number). If several active entries have the same highest priority, we nondeterministically pick one, hence abstracting away from various specific routing policies like e.g. ECMP that allow splitting a flow along multiple paths.

Definition 3. For a set of failed links $F \subseteq E$ we define the *active forwarding table* $\tau_F : V \times L \to 2^{E \times Op(L)^+}$ as $\tau_F(v, \ell) = \{(e, \omega) \mid (pr, e, \omega) \in \tau(v, \ell),\ e \in E \setminus F \text{ and } pr = pr_{min}\}$, where pr_{min} is the highest priority (minimal value) of an active forwarding entry in $\tau(v, \ell)$, or define $\tau_F(v, \ell) = \emptyset$ if $\tau(v, \ell)$ has no active forwarding entries given F.

As an example with a single protection entry, if $\tau(v_1, 01) = \{(1, e_1, \texttt{swap}(02)), (2, e_2, \texttt{swap}(02) \circ \texttt{push}(10))\}$, then given the failure scenario $F = \{e_1\}$, the cor-



(a) Network topology.



(b) Protection graph for the protections P is used for loop avoidance.

Router	Label	Prio.	e_{out}	Operation
v_1	01	1	e_1	swap(02)
v_2	05	1	e_2	swap(06)
v_3	06	1	e_3	swap(07)
v_4	02	1	out_1	pop
	07	1	out_1	pop

(c) Forwarding table, before R-MPLS, encoding flows $v_1 \to v_4$ and $v_2 \to v_3 \to v_4$.

Router	Label	Prio.	e_{out}	Operation
v_1	01	1	e_1	swap(02)
		2	lb_{v_1}	swap(02) ∘ push(10)
	10	1	e_4	swap(11)
		2	lb_{v_1}	swap(11) ∘ push(40)
	40	1	e_6	swap(41)
	31	1	e_1	pop
		2	lb_{v_1}	pop ∘ push(10)
v_2	05	1	e_2	swap(06)
	41	1	e_2	pop
v_3	06	1	e_3	swap(07)
		2	lb_{v_3}	swap(07) ∘ push(30)
	11	1	e_3	pop
		2	lb_{v_3}	pop ∘ push(30)
	30	1	e_5	swap(31)
v_4	02	1	out_1	pop
	07	1	out_1	pop

(d) Forwarding table after R-MPLS protects links e_1, e_3, and e_4 with labels (10, 11), (30, 31), resp. (40, 41). Gray rows are excluded to avoid loops.

$(in_1, 01)(e_1, 02)(out_1, \varepsilon)$ $F = \emptyset$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(e_4, 11 \circ 02)(e_3, 02)(out_1, \varepsilon)$ $F = \{e_1\}$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(lb_{v_1}, 40 \circ 11 \circ 02)(e_6, 41 \circ 11 \circ 02)(e_2, 11 \circ 02)(e_3, 02)(out_1, \varepsilon)$ $F = \{e_1, e_4\}$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(e_4, 11 \circ 02)(lb_{v_3}, 30 \circ 02)(e_5, 31 \circ 02)(lb_{v_1}, 10 \circ 02)\ldots$ $F = \{e_1, e_3\}$

(e) Traces through the network in different failures scenarios. The last looping trace (notice the repeated hop $(lb_{v_1}, 10 \circ 02)$) is avoided by excluding the grayed forwarding rules in Figure 2d.

Fig. 2: A simple network with a routing table before and after R-MPLS protection.

responding entry in the active forwarding table is $\tau_F(v_1, 01) = \{(e_2, \texttt{swap}(02) \circ \texttt{push}(10))\}$. In this case forwarding is deterministic, since $\tau_F(v_1, 01)$ is a singleton set.

Definition 4. The semantics of MPLS operations is a partial *header rewrite function* $\mathcal{H} : L^* \times Op(L)^* \rightharpoonup L^*$, where $\omega, \omega' \in Op(L)^*$, $h \in L^*$ and ε is the empty sequence of operations:

$$\mathcal{H}(h, \omega) = \begin{cases} h & \text{if } \omega = \varepsilon \\ \mathcal{H}([op](\ell) \circ h', \omega') & \text{if } \omega = op \circ \omega' \text{ and } h = \ell \circ h' \\ & \text{with } \ell \in L,\ h' \in L^* \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

where we define $[\texttt{pop}](\ell) = \varepsilon$, $[\texttt{swap}(\ell')](\ell) = \ell'$ and $[\texttt{push}(\ell')](\ell) = \ell'\ell$ for all $\ell, \ell' \in L$.

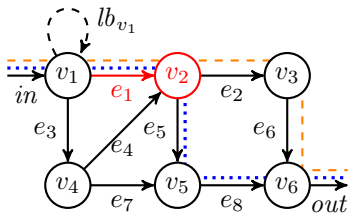
As an example, applying the operation sequence $\texttt{swap}(02) \circ \texttt{push}(10)$ to the header 01, yields $\mathcal{H}(01, \texttt{swap}(02) \circ \texttt{push}(10)) = 10 \circ 02$. The forwarding of a packet proceeds by (i) selecting an entry from the active forwarding table that corresponds to the top-most label on the packet label-stack, (ii) applying the header operations, and (iii) sending the packet on the outgoing link.

Definition 5. A *trace* in a network $N = (V, E, src, tgt, L, \tau)$, given a set of failed links $F \subseteq E$, is any (finite or infinite) sequence of link-header pairs $(e_1, h_1)(e_2, h_2)\ldots$ with each $(e_i, h_i) \in (E \setminus F) \times L^*$, where for each $i > 1$, $h_i = \mathcal{H}(h_{i-1}, \omega)$ for some $(e_i, \omega) \in \tau_F(tgt(e_{i-1}), head(h_{i-1}))$, where $head(h)$ is the top (left-most) label of h.

Figure 2e shows traces under different failure scenarios using the forwarding table in Figure 2d. The first trace is a primary path in the original data plane. The next two use R-MPLS protection in two different failure scenarios. The last one shows looping behavior, where the gray parts correspond to the gray forwarding entries in Figure 2d, which are excluded from the forwarding table by our loop avoidance algorithm.

3 R-MPLS Protection

Our recursive MPLS protection (R-MPLS) is designed as a protection layer that enhances an existing data plane. That is, it takes a topology and a data plane as inputs and returns an augmented version of the same data plane as output. This operation is performed regardless of the protocols involved in the creation of the original one. Hence our R-MPLS implementation can be used for postprocessing and data plane augmentation.



$P = \{\langle e_1, e_3e_4\rangle, \langle e_1, e_3e_7\rangle, \langle e_1, e_3e_7e_8\rangle\}$

First entries of LSP ·····
$(1, e_1, \texttt{swap}(\ell_1)) \in \tau(v_1, \ell_0)$
$(1, e_5, \texttt{swap}(\ell_2)) \in \tau(v_2, \ell_1)$

Entries for $p = \langle e_1, e_3e_7\rangle$
$(1, e_3, \texttt{swap}(\ell_p^1)) \in \tau(v_1, \ell_p^0)$
$(1, e_7, \texttt{pop}) \in \tau(v_4, \ell_p^1)$

Protect LSP ····· with p
$(2, lb_{v_1}, \texttt{swap}(\ell_2) \circ \texttt{push}(\ell_p^0)) \in \tau(v_1, \ell_0)$

Fig. 3: A network with two main forwarding paths (dotted and dashed line), and three protections P of e_1. Right side shows node protection of the dotted LSP.

We generalize the notion of link protection and node protection (Definition 6) and address which forwarding entries can be protected by a given protection path in Section 3.1. Next we solve in Section 3.2 the issue of avoiding the introduction of forwarding loops—which occurs if naively applying recursive protection. The high-level pseudocode of our protection algorithm is described in Algorithm 2 and Section 3.3, while Section 3.5 provides details on its distributed implementation.

3.1 Protectable Forwarding Entries

Definition 6. A *protection* is a pair $\langle e, p\rangle$ where $e \in E$ is the link being protected and p is a simple path $e_1 \dots e_n \in (E \setminus \{e\})^*$, with $src(e_1) = src(e)$.

If $tgt(p) = tgt(e)$, then $\langle e, p\rangle$ is a *link protection*. Figure 3 shows a network with two main LSPs (dotted and dashed lines), and three protections P for e_1. The protection $\langle e_1, e_3e_4\rangle$ is a link protection. A *node protection* routes around not just the failing link, but also the neighboring node. In the example, $\langle e_1, e_3e_7\rangle$ is a node protection. Note that this protection can only be used for the LSP going through v_5. The other LSP (going through v_3) has no node protection, but we can protect it by a path merging further down the LSP, namely the protection $\langle e_1, e_3e_7e_8\rangle$.

Not all forwarding entries can be protected by a given protection. We only install protections on forwarding entries where the original path merges with the protection path.

Definition 7. An entry $(pr, e, \omega) \in \tau(src(e), \ell)$ for a label $\ell \in L$ is *protectable* by a protection $\langle e, p\rangle$ using operations $\omega' \in Op(L)^*$, if there exists $e' \in E$ with $tgt(e') = tgt(p)$ such that for all $h \in L^*$ there is a trace

$$(e, \mathcal{H}(\ell \circ h, \omega)) \dots (e', \mathcal{H}(\ell \circ h, \omega'))$$

in the network under no failures.

The R-MPLS algorithm only installs protections for protectable entries, and it needs to know the operation sequence ω' to use. For link protection, all entries are protectable using $\omega' = \omega$.

In Figure 3, the entry $(1, e_1, \mathtt{swap}(\ell_1)) \in \tau(v_1, \ell_0)$ is protectable by $\langle e_1, e_3 e_7 \rangle$ using $\omega' = \mathtt{swap}(\ell_2)$, since $(e_1, \ell_1 \circ h)(e_5, \ell_2 \circ h)$ is a valid network trace for all $h \in L^*$. Note that applying $\omega' = \mathtt{swap}(\ell_2)$ before pushing the protection label, ensures that a packet using the protection path arrives at v_5 with the same header $\ell_2 \circ h$ as if using the main LSP.

3.2 Loop Avoidance

When using a protection $\langle e, p \rangle$, in case of a failure of a link e' on the path p, the R-MPLS algorithm allows to recursively switch to a new protection $\langle e', p' \rangle$. As shown in the last trace in Figure 2e this can sometimes result in a loop. In this example the failure of e_1 causes the packet to use the backup path $e_4 e_3$, where the failure of e_3 makes the packet switch to the backup path $e_5 e_1$, hence looping back to the failed e_1 and the backup path $e_4 e_3$.

To avoid introducing forwarding loops, we need to understand the interactions between different protection paths, and then avoid installing the recursive protection in some cases. For this, we use the following graph.

Definition 8. Given a set of protections P, we define a *protection graph* with nodes P and edges called *protection-pairs* such that there is an edge $(\langle e, p \rangle, \langle e', p' \rangle)$ in *protection-pairs* whenever the protected link e' is on the protection path p, and p' merges downstream on p, i.e. there is $e_i = e'$ and e_j with $j \geq i$ and $tgt(e_j) = tgt(p')$ such that $p = e_1 \dots e_i \dots e_j \dots e_n$. Moreover, we annotate using the function α every such edge with the set of links that must be active before the link e' is used, i.e. $\alpha(\langle e, p \rangle, \langle e', p' \rangle) = \{e_1, e_2, \dots, e_{i-1}\}$ where $p = e_1 e_2 \dots e_{i-1} e_i \dots e_n$ and $e' = e_i$.

Figure 2b gives the protection graph for the running example, when all link protections are used. The edges in the graph are labelled by their α annotations. We use the annotation to keep track of which links on the protection path p must be active, since the packet has already traversed them, when the failure of e_i makes the packet move onto the protection path p'.

Note that in our example protection graph in Figure 2b, the cycle between $\langle e_1, e_4 e_3 \rangle$ and $\langle e_3, e_5 e_1 \rangle$ corresponds to a possible forwarding loop in the failure scenario $F = \{e_1, e_3\}$ as shown by the last trace in Figure 2e. We define such bad cycles that lead to forwarding loops in some failure scenarios.

Definition 9. A *bad simple cycle* in the protection graph is a sequence of distinct protections $\langle e_1, p_1 \rangle \dots \langle e_n, p_n \rangle$ such that the set of consecutive pairs of the cycle $C \triangleq \big\{(\langle e_1, p_1 \rangle, \langle e_2, p_2 \rangle), \dots, (\langle e_{n-1}, p_{n-1} \rangle, \langle e_n, p_n \rangle), (\langle e_n, p_n \rangle, \langle e_1, p_1 \rangle)\big\}$ satisfy $C \subseteq$ *protection-pairs* and $\{e_1, \dots, e_n\} \cap (\bigcup_{(\langle e,p \rangle, \langle e',p' \rangle) \in C} \alpha(\langle e, p \rangle, \langle e', p' \rangle)) = \emptyset$.

Algorithm 1 Computing bad protection pairs to avoid loops

```
1:  function FindBadProtectionPairs(protections P)
2:      Let G be the protection graph of P (see Definition 8)
3:      B ← ∅                                   ▷ Initialize bad protection-pairs
4:      while there is a bad simple cycle of length n in G do
5:          if n > 2 then
6:              Pick an edge (⟨e, p⟩, ⟨e′, p′⟩) from the cycle
7:              Update B ← B ∪ {(⟨e, p⟩, ⟨e′, p′⟩)}
8:              Remove the edge (⟨e, p⟩, ⟨e′, p′⟩) from G
9:          else
10:             Add both protection pairs in the cycle to B
                and remove the two edges from G
11:     return B
```

The requirement in Definition 9 on the property of the cycle states that the paths in the cycle do not protect links that appear in the annotations of the protection-pairs involved in the cycle. If this property does not hold, a link is assumed to be both active and failed, hence the cycle does not correspond to a routing loop in any possible failure scenario.

Algorithm 1 removes edges, i.e. protection-pairs, from the protection graph, until there are no more bad simple cycles in the graph. The set of removed edges is returned as the bad protection-pairs, where R-MPLS should avoid adding recursive protection entries for these specific cases. We note that for cycles of length 3 and more, we only break one protection-pair on that cycle, while for shorter cycles we remove all of them (a minor optimization of the algorithm). In our example from Figure 2b, we identify the set $\{(\langle e_1, e_4e_3\rangle, \langle e_3, e_5e_1\rangle), (\langle e_3, e_5e_1\rangle, \langle e_1, e_4e_3\rangle)\}$ as the set of bad protection-pairs which form a cycle of length 2 and the links e_1 and e_3 do not appear on the annotations of the protection-pairs in the cycle.

We can find and eliminate all bad cycles using a depth-first-search approach starting from each protection, where the annotations are continuously checked to not intersect with the protected links on the search stack.

3.3 R-MPLS Algorithm

Given an existing MPLS network, with its own topology and forwarding tables, we initialize the execution of Algorithm 2 by adding *loopback* links to each router as an abstraction of instructing the router to run a packet through its forwarding processes again. The R-MPLS algorithm then installs the LSP for each protection (loop in Lines 4–10). We here give the protections as input to the algorithm. They can be computed e.g. as link or node protections along shortest paths. Each router along the protection allocates a local label $\ell^i_{\langle e,p\rangle}$ to

Algorithm 2 Recursive protection algorithm

Input: Network $N = (V, E, src, tgt, L, \tau)$, set of protections $P \subseteq (E \times E^*)$
Output: Protected network $N' = (V, E', src', tgt', L', \tau')$ with R-MPLS protection

1: $src' \leftarrow src$, $tgt' \leftarrow tgt$, $L' \leftarrow L$, $\tau' \leftarrow \tau$ ▷ *Initialize* N'
2: **for** $v \in V$ **do** create loopback link lb_v such that $src'(lb_v) = v$ and $tgt'(lb_v) = v$
3: $E' \leftarrow E \cup \{lb_v \mid v \in V\}$
4: **for** each protection $\langle e, p \rangle \in P$ **do** (let $e_1, \ldots, e_n$ be the links on protection path p)
5: $e_0 \leftarrow lb_{src(e)}$ ▷ *Start from loopback link*
6: $\ell^0_{\langle e,p \rangle}, \ldots, \ell^{n-1}_{\langle e,p \rangle} \leftarrow$ fresh labels, add each $\ell^i_{\langle e,p \rangle}$ to L'
7: $protects(\ell^i_{\langle e,p \rangle}) \leftarrow \langle e, p \rangle$ for $0 \leq i < n$ ▷ *Store in mapping protects* : $(L' \setminus L) \to P$
8: **for** $i \in \{0, 1, \ldots, n-2\}$ **do**
9: $\tau'(tgt(e_i), \ell^i_{\langle e,p \rangle}) \leftarrow \{(1, e_{i+1}, \mathtt{swap}(\ell^{i+1}_{\langle e,p \rangle}))\}$ ▷ *Use labels to encode the path*
10: $\tau'(tgt(e_{n-1}), \ell^{n-1}_{\langle e,p \rangle}) \leftarrow \{(1, e_n, \mathtt{pop})\}$ ▷ *Pop the label on last hop*
11: *bad-protection-pairs* $\leftarrow$ FindBadProtectionPairs(P) ▷ *Call Algorithm 1*
12: Let M be larger than any priority occurring in τ'
13: Let $\tau'_{min}(v, \ell) = \{(pr, e, \omega) \in \tau'(v, \ell) \mid pr = pr_{min}\}$ where pr_{min} is the highest priority in $\tau'(v, \ell)$
14: **for** $v \in V, \ell \in L', (pr, e, \omega) \in \tau'_{min}(v, \ell)$ and $\langle e, p \rangle \in P$ **do**
15: **if** $\ell \in L$ **then** ▷ *Protection of original data plane*
16: **if** entry $(pr, e, \omega) \in \tau'(v, \ell)$ is protectable by $\langle e, p \rangle$ using $\omega' \in Op(L)^*$ **then**
17: $\tau'(v, \ell) \leftarrow \tau'(v, \ell) \cup \{(M,\ lb_v,\ \omega' \circ \mathtt{push}(\ell^0_{\langle e,p \rangle}))\}$ ▷ *Push backup path*
18: **else** (let $\langle e', p' \rangle = protects(\ell)$, let $e'_1, \ldots, e'_n$ be the links on p', and let j be the index where $tgt(e'_j) = v$)
19: **if** $(\langle e', p' \rangle, \langle e, p \rangle) \notin$ *bad-protection-pairs* **and** there exists index i such that $tgt(e'_i) = tgt(p)$ and $i > j$ **then**
20: $\omega' \leftarrow \begin{cases} \mathtt{pop} & \text{if } i = n \\ \mathtt{swap}(\ell^i_{\langle e',p' \rangle}) & \text{otherwise} \end{cases}$
21: $\tau'(v, \ell) \leftarrow \tau'(v, \ell) \cup \{(M,\ lb_v,\ \omega' \circ \mathtt{push}(\ell^0_{\langle e,p \rangle}))\}$ ▷ *Recursive protection*
22: **return** $N' = (V, E', src', tgt', L', \tau')$

it and records that the label is on the LSP of protection $\langle e, p \rangle$. It then creates new entries in its forwarding table to use the protection path. Notice that the last router in the path uses a pop instruction while the others just swap labels. No router along this path records a push instruction, as these LSPs are only used as protection paths.

Next, we compute a set of bad protection-pairs based on the protection graph using Algorithm 1. In order to avoid introducing forwarding loops, we disable the recursive protection for these specific pairs of protections.

After completing the previous process, each router proceeds to execute the loop in Lines 14–21 which augments the original forwarding table by adding lower priority entries. Note that this loop does not iterate over the new entries created inside the loop. For each previously existing highest priority

forwarding entry, and for each protection that can protect that forwarding entry (Line 16), the router creates a new lower priority entry (Line 17). The new entry performs the operations ω' that will make the packet arrive at the merge point router with the same header as under the original forwarding. These operations are followed by pushing the label that encodes the protection path. The new protection entries forward to the router's loopback link. For link protection, the sequence ω' is just the original operations ω for the entry; for other protections this information needs to be retrieved from e.g. the control plane.

To achieve recursive protection, the same is done for the entries created on the loop of Lines 4–10, unless the protection that the incoming label encodes, paired with the protection we are about to use, are part of the bad protection-pairs. Again, we check if the entry is protectable by the protection, i.e. that the new protection intersects downstream with the current protection. The operation ω' is computed based on where the two protection paths intersect.

We apply Algorithm 2 on the network topology from Figure 2a and forwarding table in Figure 2c that encodes two flows. We use the link protections P in Figure 2b. The resulting forwarding table is shown in Figure 2d, where the links e_1, e_3, and e_4 are protected. Note that in this small example, no other links can have link protection due to the direction of the links; however, in real networks usually all links get protection paths. Figure 2e shows four possible traces under different failure scenarios. Notice how the third trace uses the recursive protection to recover from both e_1 and e_4 failing. Due to the failure of e_1, it first tries to use the protection path starting at e_4 encoded by the label 10. Using the loopback link, it then tests if link e_4 is also failing, and then it uses the path through e_6 until that path joins the first protection path at router v_3.

3.4 Recursive Link and Node Protection

Algorithm 2 takes the set of protections P as input. We now show two instantiations of computing P: link protection and node protection.

Given a network topology (V, E, src, tgt) where the links are annotated with weights, we compute for each link $e \in E$ the shortest path p from $src(e)$ to $tgt(e)$ in the graph $(V, E \setminus \{e\}, src, tgt)$, and if p exists add $\langle e, p \rangle$ to the set of protections P. This gives link protection of all links.

To compute node protection: for each $v \in V$ and $e, e' \in E$ with $tgt(e) = v = src(e')$, compute the shortest path p from $src(e)$ to $tgt(e')$ in the graph with v and all of v's incident edges removed. If p exists, add $\langle e, p \rangle$ to the set of protections P.

The standard FRR facility protection uses node protection when possible and link protection only as a fallback. We can achieve the recursive version of this, by adding both link and node protections to P, and then extending

Line 16 and 19 to filter out link protections in case the entry is protectable by a node protection. The R-MPLS framework also allows for more general sets of protections, e.g. to optimize link capacity usage in failure scenarios.

3.5 Distributed R-MPLS Implementation

The pseudocode from Algorithm 2 and the computation of protections P for common protection schemes like link and node protection, can be implemented in a fully distributed fashion, and it is hence compatible with traditional MPLS routers. In particular, each router can compute the protection paths for each of its outgoing links. The topology knowledge required to compute paths is provided on traditional MPLS networks by the Interior Gateway Protocol (IGP), typically OSPF-TE [23] or ISIS-TE [24]. The information exchanged by the IGP is stored by each router in a local database, so no central entity with a complete view of the topology is required. Each router along the computed protection path is notified by the originating router, and then Lines 4–10 of Algorithm 2 are executed locally.

Notice that Line 11 and Line 18 (with the mapping defined on Line 7) require knowledge of the full protection path $p = e_1, \ldots, e_n$. To obtain this information, the intermediate routers along the path $src(e_2), \ldots, src(e_n)$ query $src(e_1)$, which is responsible for computing the protection path. Such a query can be performed e.g. using the RSVP Diagnose facility [25], by which any network element sends a request message to another router and inquires information about computed paths. This request uses only existing RSVP primitives so the communication can be implemented completely in software.

The loop on Lines 14–21 of Algorithm 2 requires only local operations on each router, when link protection is used. For node protections, Line 20 queries the merge point router for its label allocated to the protection, and Line 16 needs to query the next-hop router $tgt(e)$ for its forwarding entries of the top label left by ω. In other words, each router only needs to know about the labels of neighboring routers to implement link protection and their next-hop forwarding entries to implement node protection. No other information nor central controller is needed.

R-MPLS has then all the information available, when it finishes computing its protection paths. So, whichever path is provided by the underlying protocols, at present or in the future, as long as R-MPLS finished computing its own (topology dependent only) protection paths, then the router can derive a protecting entry for each original data plane route, loop-freedom guaranteed; (though for node protections, Line 16 still needs to query the neighboring router to get ω'). And if or when new forwarding paths result from the underlying protocols, these can also be protected against link failures. Since the priority range is partitioned, all R-MPLS routing entries are guaranteed to have lower priority than all original ones, ensuring no interference on the networks'

basic routing.

Execution of Algorithm 1 can also be implemented in a distributed fashion. Again, each router only performs computations for its own outgoing links. Routers query each other for the protections paths they have previously computed. The rest of the algorithm is computed locally from those elements. To ensure that computations made on Line 6 are identical on all routers, it suffices to assume a total order on the set of links and choose the protection-pair deterministically regardless of the router.

3.6 Properties of the R-MPLS Protection

We shall now argue that our R-MPLS protection preserves all the connections of the original MPLS data plane and does not introduce any forwarding loops. For this we first need to define the subset of traces that corresponds to a full run of a packet.

Definition 10. Let $N = (V, E, src, tgt, L, \tau)$ be an MPLS network and let $F \subseteq E$ be the set of failed links. A *maximum trace* in N under F is either any infinite trace or a finite trace that is not a prefix of any other trace in N under F.

Hence a finite trace $(e_1, h_1) \dots (e_n, h_n)$ is maximum if its last link-header pair (e_n, h_n) satisfies either $h_n = \varepsilon$, $\tau_F(tgt(e_n), head(h_n)) = \emptyset$, or $\mathcal{H}(h_n, \omega)$ is undefined for all $(e, \omega) \in \tau_F(tgt(e_n), head(h_n))$.

We now formally define the three properties of MPLS networks. Due to the support of nondeterministic forwarding, there is a difference between possibility and certainty of connectivity in a given scenario.

Definition 11. For a network $N = (V, E, src, tgt, L, \tau)$ and set of failed links $F \subseteq E$, define the predicates *no-loops*$_N^F$, *can-reach*$_N^F$, and *must-reach*$_N^F$ such that for $e, e' \in E$ and $h, h' \in L^*$:

- *can-reach*$_N^F(e, h, e', h')$ is true iff there exists a trace $(e, h) \dots (e', h')$ in N under F,
- *must-reach*$_N^F(e, h, e', h')$ is true iff every maximum trace starting at (e, h) contains (e', h'), and
- *no-loops*$_N^F$ is true iff every maximum trace in N under F is finite.

We now show that Algorithm 2 preserves all *can-reach* and *must-reach* properties, i.e. our protection never removes connectivity. We refer to the appendix for the proofs of the theorems.

Theorem 1. *Let N' be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. For all possible failure scenarios $F \subseteq E$, for all $e, e' \in E$ and $h, h' \in L^*$:*

(1) if can-reach$^F_N(e, h, e', h')$ *then can-reach*$^F_{N'}(e, h, e', h')$,

(2) if must-reach$^F_N(e, h, e', h')$ *then must-reach*$^F_{N'}(e, h, e', h')$.

The next theorem states that for any loop-free input data plane, R-MPLS guarantees to produce a loop-free protected data plane.

Theorem 2. *Let* $N' = (V, E', src', tgt', L', \tau')$ *be the result of applying Algorithm 2 for recursive protection to an MPLS network* $N = (V, E, src, tgt, L, \tau)$. *If for all failure scenarios* $F \subseteq E$ *the network* N *satisfies no-loops*^F_N *then for all failure scenarios* $F' \subseteq E$ *the protected network also satisfies no-loops*$^{F'}_{N'}$.

4 Evaluation of R-MPLS

In this section we describe the experimental evaluation of R-MPLS. We compare its protection performance as well as memory and communication overhead against the unprotected data plane, the industry standard FRR protection and the optimal protection achieved by the tool Plinko [26].

4.1 MPLS Generation and Simulation

For evaluation of MPLS data planes we use MPLS-Kit [27], a tool and library for data plane generation and simulation. It includes utilities for automation of execution and analysis. Specifically, MPLS-Kit provides two main functionalities:

- **Data Plane Generation.** Allows for the computation of the converged data plane, mimicking a real network by running the industry standard control protocols Label Distribution Protocol (LDP) [28] and the Resource ReSerVation Protocol (RSVP) [18]. It does so by exchanging the same information that these protocols do in real networks, yet without engaging in a simulation of the actual message passing. The user inputs the network topology and the required control protocol parameters. With the information at hand, each router in the topology allocates MPLS labels and populates its forwarding tables accordingly to provide the intended reachability.
- **Simulation.** Once the data plane is generated, the library also provides functionality to perform simple packet-level simulations in order to test reachability. For this purpose, packet-level simulators model and mimic the packet delivery through the network on a hop by hop basis. The resulting trajectory of the packet along the network serves as a witness in testing that the packet arrives to its intended destination. In particular,

our simulator initializes an MPLS packet with a valid header to be handled by routers in the topology. The packet is then forwarded according to the data plane rules until there are no more labels on the header or the time-to-live field is exceeded (i.e, a forwarding loop).

Further details regarding both core functionalities are provided in the appendix. The code, dataset and experimental setup is publicly available as an artifact [22], and details on the artifact are given in the artifact appendix.

4.2 Methodology

To empirically evaluate the reliability achieved by our proposed recursive protection method, we perform a series of experiments that can be decomposed in two sets:

- **RSVP experiments**: Adding R-MPLS protection on top of RSVP-based data planes.
- **LDP experiments**: Adding R-MPLS protection on top of LDP-based data planes.

The topologies used as input for MPLS-Kit [27] are real-world networks from the topology Zoo dataset [29]. For the topologies in the dataset, we first generate data planes and then we enumerate all failure scenarios (sets of failed links) with up to 4 failed links. Then, for each combination of topology, data plane and failure scenario, we run a set of packet-level simulations for all labels representing valid user traffic. As the LDP data plane is nondeterministic, we run here multiple packet simulations and take the average.

Data Plane Generation

On RSVP experiments, we compute the following data planes (referred as *RSVP-based data planes*):

- **RSVP:** Data plane containing n^2 unprotected RSVP tunnels between random endpoints, where n is the number of nodes.
- **RSVP + R-MPLS {Link,Node}:** Data plane containing the exact same tunnels as RSVP and additional R-MPLS recursive protection on top with either single link protections (Link) or node protections with link protection as fallback (Node).
- **RSVP + FRR:** Data plane containing the exact same tunnels as RSVP and additional RSVP-TE Fast Reroute node protection [4].

- **RSVP + Plinko {2,4}:** Data planes containing the exact same tunnels as RSVP and additional Plinko [26] path protection, with resiliency levels 2 and 4.

This set of six data planes per topology allows for direct evaluation of the impact of adding R-MPLS on top of an unprotected RSVP data plane, and to compare against the benchmarks of industry-standard RSVP-TE FRR and the state-of-the-art high-resiliency approach of Plinko.

RSVP-TE FRR is widely used on real MPLS networks to provide temporary protection against networking failures. It has been in use for more than a decade and is well understood. It is designed to provide sub 50ms local responses by diverting traffic in case of link failure and comes in two modes, facility (node) protection where the precomputed protection path avoids sending traffic through the next hop of the failed link, and link protection where other links to the same next hop can be used.

Plinko is a state-of-the-art technique for achieving optimal resiliency, i.e., it provides protection to existing routes on scenarios of up to t link failures, as long as there exists a path in the topology, without introducing loops. Plinko can be implemented using RSVP-TE FRR primitives, so we extended the RSVP class of MPLS-Kit to provide protection as specified by Algorithm 1 in [26]. A drawback hindering widespread adoption of Plinko on real networks is its high memory consumption. The original paper proposes a non-MPLS forwarding model that allows a reduction of required storage space but this improvement is impossible to employ on traditional networking devices. In our experiments, we use Plinko with resiliency levels 2 and 4. This means that for value 4, Plinko computes a protection path for up to 4 failed links (provided that the topology remains connected).

On LDP experiments, we compute just two data planes (referred as *LDP-based data planes*):

- **LDP** Data plane containing LDP generated labels for reaching every link and node in the topology from all routers.
- **LDP + R-MPLS {Link,Node}** Same data plane as LDP but with additional R-MPLS recursive protection on top with either link protection (Link) or node protection with link protection as fallback (Node).

In this case, we evaluate the effect of adding the recursive protection on top of an unprotected data plane. Notice that as FRR is a RSVP-TE specific protection mechanism, it cannot be applied to LDP.

Failure Scenarios

We generate the failure scenarios by choosing all possible combinations of k links on all the topologies, for all values of k between 0 and 4 (included). This

means that all failure scenarios have a number of total failed edges of at most 4. When we generate a failure on a topology link, we remove both directed links between the corresponding nodes, thus provoking a disconnection in both directions. This is usually the case in real networks. To restrict the total number of cases to test, if the number of total combinations to evaluate is larger than $\binom{40}{4}$, we randomly choose that number of scenarios from the set of all possible ones with at most 4 failed links. In this case no distinction is made between LDP and RSVP experiments. Notice that when available, Plinko with resiliency level 4 provides protection for the maximum number of failed links we consider. This implies that Plinko (level 4) always achieves the optimal protection level in our simulations, however, at the expense of exponentially large communication and memory overheads as demonstrated by our experiments.

Execution Details

All simulations are executed making use of the command line tools and the library we implemented. In order to evaluate the topology as an MPLS network, we only simulate packets that can be part of user-generated traffic. A key advantage that this method provides, is that it allows to test the exact same packets on all data planes for each kind of experiment (RSVP or LDP), simplifying the comparison of the results.

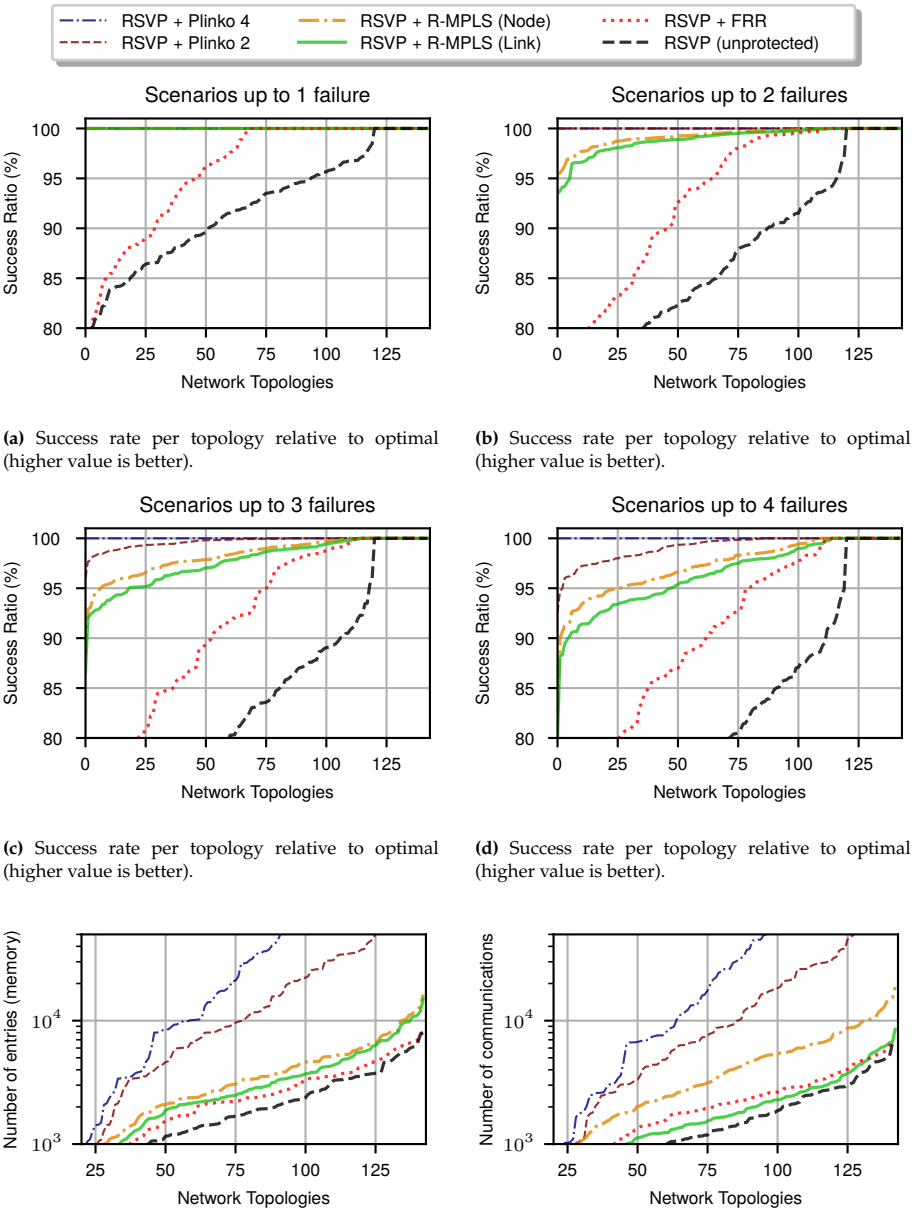
During the simulations we also count how many times the simulation of a packet forwarding ends up in a successful forwarding towards its intended destination and how many times it ends in a failure. Given the large number of experiments, we execute the experiments on a compute cluster with 9 machines, each with 64 cores. We conduct the experiments on all topologies with a single connected component of up to 40 links.

The size of the topologies in our experiments is constrained by the computation times required to compute Plinko and LDP data planes. These two protocols have poor space scalability (exponential for Plinko and quadratic for LDP), therefore using larger topologies leads to excesively large delays in the tool MPLS-Kit [27] just to obtain the baselines, without contributing significantly to the results. This criterion results in a subset of 143 topologies.

4.3 Results of RSVP Experiments

In our experiments, we analyze the success rates of the protection, memory overhead (related to the number of added protection rules) and communication overhead (number of messages needed to establish the protection in a distributed way).

From the experience using our data plane generation tool, on the large majority of topologies the time required to compute R-MPLS entries is com-



(a) Success rate per topology relative to optimal (higher value is better).

(b) Success rate per topology relative to optimal (higher value is better).

(c) Success rate per topology relative to optimal (higher value is better).

(d) Success rate per topology relative to optimal (higher value is better).

(e) Number of total forwarding table entries per topology (lower value is better).

(f) Number of communication messages among nodes (lower value is better).

Fig. 4: Results for R-MPLS on all RSVP based data planes. Note that in (e) and (f) the y-axis is logarithmic.

Topology	#nodes	#links	RSVP	FRR	R-MPLS (Link)	R-MPLS (Node)	Plinko 2	Plinko 4
Ans	18	25	61 %	70 %	87 %	88 %	90 %	94 %
Heanet	7	11	55 %	77 %	80 %	83 %	85 %	88 %
Uninet	13	18	58 %	81 %	83 %	84 %	87 %	88 %
EliBackbone	20	30	68 %	85 %	93 %	94 %	96 %	98 %
Abvt	23	31	65 %	74 %	90 %	91 %	93 %	95 %
Cesnet200304	29	33	69 %	80 %	82 %	82 %	83 %	83 %
Nextgen	17	19	44 %	48 %	56 %	58 %	60 %	60 %
Harnet	21	23	59 %	71 %	71 %	71 %	73 %	74 %
Getnet	7	8	39 %	50 %	51 %	52 %	54 %	54 %
GtsRomania	21	24	64 %	72 %	76 %	76 %	77 %	77 %
Nordu1997	14	13	49 %	49 %	49 %	49 %	49 %	49 %
Arn	30	29	67 %	67 %	67 %	67 %	67 %	67 %
Reuna	37	36	57 %	57 %	57 %	57 %	57 %	57 %
Amres	25	24	46 %	46 %	46 %	46 %	46 %	46 %
Basnet	7	6	31 %	31 %	31 %	31 %	31 %	31 %

Table 1: Success rates for topologies using different protections ordered by the improvement R-MPLS (Link) gives on unprotected RSVP. The table shows the five top, five middle and five bottom rows.

parable to the time to compute RSVP+FRR entries. Additionally, the observed average number of additional hops is almost identical between RMPLS and RSVP+FRR protections.

Success Rates

Figures 4a-4d show, for increasing numbers of failed links, plots for the success rates achieved by each RSVP-based data plane on each topology. Results are averaged over all failure scenarios considered, providing a measurement on the fraction of successful cases for each network topology and the network topologies are sorted in non-decreasing order (on x-axis) according to their success rates (y-axis). We can observe that Plinko 4 indeed provides optimal protection upto 4 link failures, and Plinko 2 up to two link failures. Our R-MPLS provides perfect protection for 1 link failure and slighly deteriorates with the increasing number of failures. In the rest we focus on the discussion of Figure 4d, which contains all the simulated scenarios with up to 4 link failures.

Clearly, the unprotected RSVP has the smallest success rates, and all protected data planes achieve higher success rates. The standard FRR node protection on top of RSVP achieves, as expected, a considerably better success rate. Adding our R-MPLS protection on top of the unprotected RSVP data plane alone clearly outperforms RSVP with the standard FRR node protection, both for the link and node protection. This is because RSVP+FRR has only one

option to provide a protection while R-MPLS adds recursive (multiple edge) protection. Our R-MPLS protections gets closer in success rate to the optimal protection achieved by Plinko level 4 (protection is guaranteed anytime there is physical connectivity), while Plinko's level 2 success rate is between R-MPLS and the optimal protection. As expected, the node protection R-MPLS achieves better success ratio than the link protection. In all experiments, as formally proved earlier, we confirm that R-MPLS does not create any forwarding loops.

Table 1 highlights the 5 topologies (from the tested 143) that achieved the largest improvement by adding R-MPLS protection on top of the unprotected RSVP data plane, as well as the 5 in the middle and the 5 topologies with the lowest improvement. It also provides further details about the size of the topologies. Note that the 5 bottom topologies are trees, and hence cannot be protected against link failures by any method. For the top 5 topologies in the table, large improvements are achieved by all protection schemes. Yet clear differences are present between the R-MPLS protected data planes and FRR: at least 2% for link protection (3% for node protection) and up to 17% (resp 18%) on the first 5 topologies. On many occasions, the R-MPLS solutions get closer to the optimal value achieved by Plinko (at level 4) than to the standard FRR.

Memory and Communication Overhead

Figure 4e shows the accumulated number of entries in the forwarding tables (corresponding to the required memory) of the routers, where the topologies are sorted (on the x-axis) according to the number of entries (y-axis). We can see that both FRR and our R-MPLS approach add only a moderate number of additional forwarding rules to the existing data plane (with only small differences between node and link protection). On average 21% of the memory on FRR protected data planes are used for the protection, where for R-MPLS the number is 35% and 44% for link and node protection respectively. Plinko protects (whenever possible) against all failure scenarios with up to 4 failed links, but it requires exponentially many more entries in the forwarding tables to do so (note that the y-axis is logarithmic). This is also the case if we consider Plinko only at level 2; now Plinko does not provide the optimal protection for 4 link failures anymore but at the same time it still has an exponential overhead for establishing the protection.

Similarly, Figure 4f shows the amount of required communication (message exchanges between the nodes) for all considered network topologies, showing only a negligible overhead for establishing FRR and R-MPLS link protection (on average 31% resp. 23% of the communications are used for protection) but a large communication penalty for adding the optimal protection by Plinko, both for level 4 and 2. We also notice that computing the R-MPLS node protection requires larger number of communications (on average 60% of the communications) compared to the link protection. This is due to the fact that

routers must query the neighboring routers about the labels used for encoding downstream header rewriting.

As a conclusion, the memory overhead to establish our R-MPLS protection in a distributed way is small and comparable to the widely used FRR protection, however, the success rate of the R-MPLS protection is significantly higher than for FRR. R-MPLS link protection requires fewer message exchanges between the routers compared to the node protection. Plinko achieves the optimal success rate, however, at the expense of unrealistic demands on the available memory and with a large communication overhead.

4.4 Results of LDP Experiments

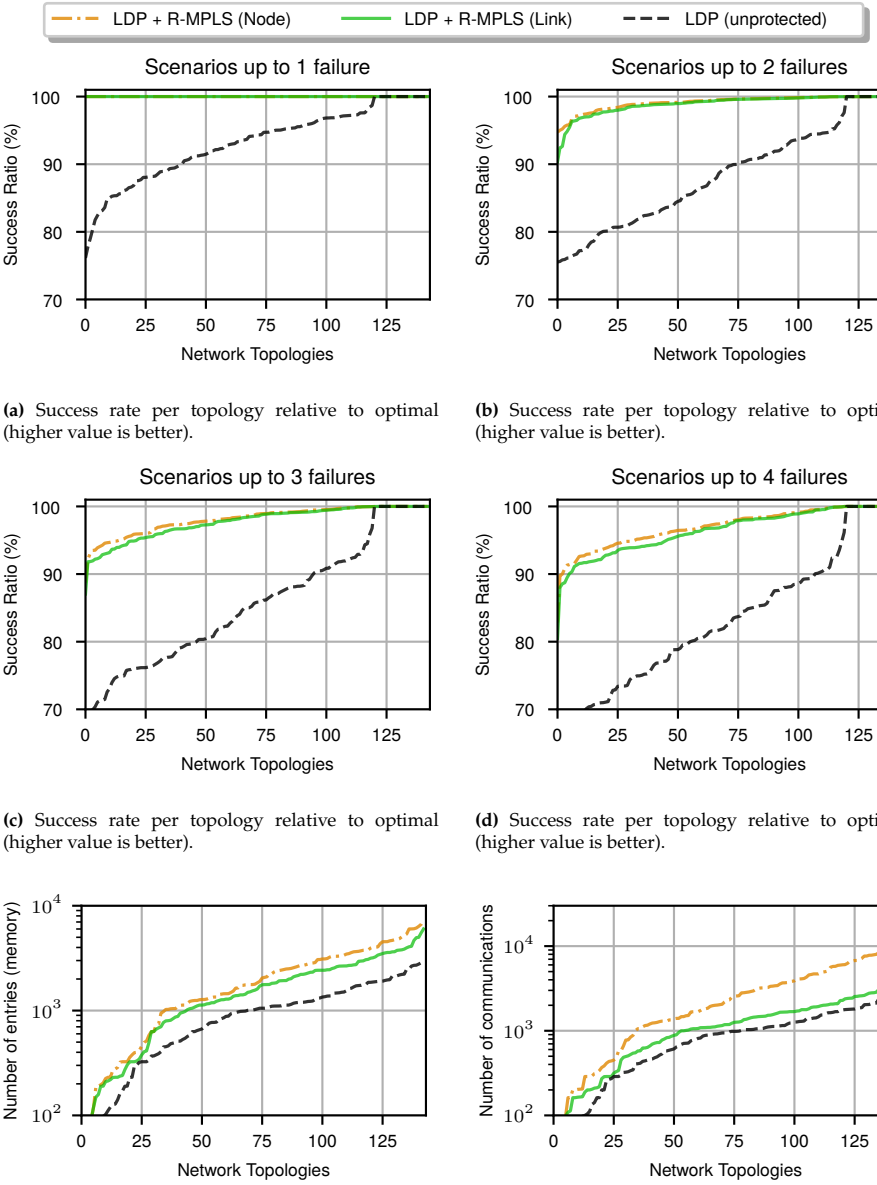
We first focus on the success rate achieved by both protected (R-MPLS) and unprotected LDP data planes and benchmark against the optimum which indicates a success if the failure scenario still allows at least one path from source to destination. We then consider the memory and communication overhead. As discussed earlier, FFR and Plinko are not applicable for protecting an LDP data plane.

Success Rates.

Figures 5a-5d show, for increasing numbers of failed links, plots with sorted success rates achieved by the LDP-based data plane and its R-MPLS protection relative to the optimum achievable protection. As before, we can observe that R-MPLS protects optimally up to 1 link failure and with the increasing number of link failures, it provides significant improvement over the unprotected data plane. The curves in Figure 5d confirm the observations from the RSVP experiments, showing significantly improved success rates when the basic LDP data plane is protected using R-MPLS and we are also relatively close to the optimum protection. As before, R-MPLS node protection is slightly more successful than the link protection.

Memory and Communication Overhead

The plots in Figure 5e and 5f follow the same trend as for protection of RSVP data plane and show that the overhead both for the number of entries in the forwarding tables and in the communication overhead is moderate and proportional to the overhead for establishing the unprotected data plane using LDP. Out of all created forwarding rules, only 39% (on average) are used for the additional recursive link protection and 49% for node protection, with about 27% of message exchanges needed to establish the link protection and 61% for node protection.



(a) Success rate per topology relative to optimal (higher value is better).

(b) Success rate per topology relative to optimal (higher value is better).

(c) Success rate per topology relative to optimal (higher value is better).

(d) Success rate per topology relative to optimal (higher value is better).

(e) Number of total forwarding table entries per topology (lower value is better).

(f) Number of communication messages among nodes (lower value is better).

Fig. 5: Results for R-MPLS on LDP based data planes. Note that in (e) and (f) the y-axis is logarithmic.

5 Discussion

Our R-MPLS protection is designed for working on top of an arbitrary MPLS data plane. To realize this efficiently we have to address the issue of packet recirculation. Our solution uses the logical loopback link to check link failures one at a time. In case of failures, this induces a runtime overhead that is linear in the number of failed links at the router. The advantage, however, is that only one path needs to be added per link we protect, and it only takes one entry to add recursive protection for each existing entry, so the memory overhead is minimal. An alternative approach is to compute protection paths for each router and for each subset of its links that can fail. With this approach, there is no time overhead, however, an explosion in the number of necessary entries in the forwarding tables of the routers.

In our R-MPLS implementation, we hence resort to packet recirculation (where packets are sent to the loopback interface) inside the router to provide resiliency against failures. Yet, recirculating packets requires these to be sent through a slow processing path to a control element that introduces the packet again into forwarding hardware. This can hurt the throughput and cause a spike in the router's CPU. To avoid these effects, we propose a further R-MPLS enhancement without altering its inner working by recirculating the first few packets of a flow after an adjacent link failure and caching the set of header operations and the outgoing interface through which the packet finally gets transmitted. This information is then used to insert a new temporary routing entry in the forwarding table, with a priority such that it matches the following packets of the same flow. The new entry is valid until the router detects a new local link up or link down event. This operation avoids further recirculation of packets while preserving the same protection intended by R-MPLS. We plan to develop this concept in future work. In the case of non-traditional MPLS devices, it is possible to implement this caching mechanism e.g. using PURR [30], a technique devised specifically to provide packet recirculation-free primitives for path protections on programmable routers.

Although requiring fewer routing entries, R-MPLS may result in deep label stacks in multiple failure scenarios, leading to potential fragmentation or maximum label depth issues. The former can be alleviated with jumbo frames [31] without requiring lowering the MTU, and the latter with label replacement techniques in which a label stack is replaced by a new, shallower stack.

6 Related Work

To provide high availability in the presence of failures, most modern communication networks support fast recovery in the data plane [7, 18, 32], see [4] for a recent survey.

This paper focuses on conventional MPLS networks, which are widely deployed today. Compared to alternative network types [33–37], a particular property and challenge of MPLS networks is that the header size is dynamic and potentially unbounded. The ability to fast reroute traffic (i.e., to protect LSPs) is a key feature of MPLS [16–18]. Most work has, however, been on single failure protection techniques, e.g, RSVP-TE FRR [18], LFA [38] and TI-LFA, [39]. Limitations of these techniques in multi-failure scenarios have already been observed [40–47]. Besides RSVP-TE FRR, which has already been discussed in this article, LFA is a solution LDP Fast ReRoute. LFA requires knowledge of the paths to destination, so it cannot be used independently of its specific control protocol, while R-MPLS works for any control protocol—it uses the LFIB entries but is not concerned about how they were generated. Hence, R-MPLS does not interact with any other control protocol. Additionally, both LFA and TI-LFA have been designed to protect against a single failure, while R-MPLS is more general, so it is expected that R-MPLS outperforms both on multi-link failure scenarios. Alternatively, one can consider the resilience provided by Equal Cost Multi-Path (ECMP), a load-balancing data plane mechanism. As ECMP is not a Fast ReRoute protection scheme, it cannot be directly directly compared with R-MPLS. Furthermore, our model and simulator support ECMP that is abstracted as nondeterministic forwarding.

The approach suggested in [7] runs a data plane re-convergence algorithm by reversing the directions of links upon failures, while modifying the routing tables. This approach is orthogonal to ours as we preinstall the failover directly in the data plane.

Although there are proposals for achieving forwarding resilience up to a maximum number of link failures that do not disconnect the topology (*perfect forwarding resiliency* [48]) on top of MPLS primitives, we are not aware of a solution that achieves such resilience in a conventional MPLS network. Some existing proposals, like R3 [40] have a mandatory centralized stage and require additional traffic demand information, which is usually not available. R-MPLS achieves such resilience while being fully distributed and not requiring external information.

Protecting the protection paths is mentioned in RFC6981 [49], where the issue of mutually looping protection paths is addressed by putting such links into a secondary shared risk link group (SRLG), but—to our knowledge—this has not been implemented. Compared to [49], we extend with multiple protection paths, provide a complete algorithm for eliminating loops and support node protection. Further, we implement our algorithm along side with existing protocols, and run experiments to compare the performance.

R-MPLS is attractive for its ability to reinforce an existing forwarding data plane independently of how it was built. This is in stark contrast with Plinko [26], which is the only state-of-the-art proposal we know of capable of achieving perfect forwarding resilience that can be applied to conventional

MPLS. However, Plinko requires control plane knowledge, i.e. the information on how the forwarding paths were originally computed, and thus cannot be easily integrated with traditional MPLS control protocols. Moreover, Plinko brute-force enumerates all hypothetical failure scenarious that must be encoded into (exponentially large) label space, causing a combinatorial number of inserted forwarding entries and exchanged messages among the routers. Our R-MPLS does not introduce this explosion in the number of labels and rules and hence scales memory- and communication-wise better than Plinko, whereas Plinko on the other hand achieves better connectivity. The requirement on the knowledge of the control plane also affects *Failure Carrying Packets* (FCP) [50], a classical proposal similar to Plinko.

Recent work showed how to provably verify the resilience and policy-compliance of MPLS networks under multiple failures. In particular, tools such as P-Rex [19, 51] and AalWines [20] allow verifying the reachability of MPLS data planes even under failures in polynomial time. However, in contrast to R-MPLS, these approaches cannot be used to improve the resilience of the data plane.

Last but not least, our work is orthogonal to solutions such as PURR [30], which allows to avoid overheads of recirculation in the switch during failover.

7 Conclusion

Motivated by uncovering the opportunity to increase the resilience of MPLS networks, we suggest a recursive MPLS data plane protection, allowing us to provably route traffic around *multiple* simultaneously failed links without creating any forwarding loops. Contrary to other existing approaches, R-MPLS is *fully distributed* solution and hence it is compatible with existing MPLS hardware employed in current networks.

We evaluate R-MPLS on protecting real-world networks with realistic data planes and show that our approach is efficient and significantly increases network robustness compared to the state-of-the-art FRR protection, at similar memory and communications cost. Another feature of our solution is that it is orthogonal and can be combined with existing and future protocols, such as RSVP or LDP, serving as an "extra resilience" layer, while requiring only minimal increase in memory and communication overhead.

Our work opens several interesting directions for future research. We plan to extend R-MPLS to Segment Routing networks and to evaluate its performance with respect to the standard Segment Routing's protection TI-LFA. Also, we plan to study how to further improve the performance of our algorithms. Our approach is also readily available to account for link congestion when fast reroute takes over because in our protection algorithm, we can select an arbitrary protection path for a link.

Acknowledgements. This research is funded by the Vienna Science and Technology Fund (WWTF), project WHATIF (ICT19-045), and DFF project QASNET.

References

[1] N. Shelly, B. Tschaen, K.-T. Förster, M. Chang, T. Benson, and L. Vanbever, "Destroying networks for fun (and profit)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015, article 6, pp. 1–7.

[2] C. Labovitz, G. R. Malan, and F. Jahanian, "Internet routing instability," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 515–528, 1998.

[3] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM*, 2011, pp. 350–361.

[4] M. Chiesa, A. Kamisinski, J. Rak, G. Retvari, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys and Tutorials (COMST)*, pp. 1253–1301, 2021.

[5] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *Proc. USENIX NSDI*, 2019, pp. 161–176.

[6] K.-T. Foerster, J. Hirvonen, Y.-A. Pignolet, S. Schmid, and G. Tredan, "On the feasibility of perfect resilience with local fast failover," in *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

[7] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *Proc. USENIX NSDI*, 2013, pp. 113–126.

[8] M. Chiesa, I. Nikolaevskiy, S. Mitrović, A. Panda, A. Gurtov, A. Maidry, M. Schapira, and S. Shenker, "The quest for resilient (static) forwarding tables," in *Proc. IEEE INFOCOM*, 2016.

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

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

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

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

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

[14] A. Atlas and A. D. Zinin, "Basic specification for IP fast reroute: Loop-free alternates," RFC 5286, Sep. 2008.

[15] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "IP fast rerouting for multi-link failures," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 3014–3025, 2016.

[16] S. Smith, *Introduction to MPLS*, https://www.cisco.com/c/dam/global/fr_ca/training-events/pdfs/Intro_to_mpls.pdf, 2003, visited: 19/05/2020.

[17] E. C. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," RFC 3031, Jan. 2001.

[18] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," RFC 4090, May 2005.

[19] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[20] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[21] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[22] S. Schmid, M. K. Schou, J. Srba, and J. Vanerio, "Artifact for "R-MPLS: Recursive Protection for Highly Dependable MPLS Networks"," Zenodo, Oct. 2022.

[23] D. M. Yeung, D. Katz, and K. Kompella, "Traffic engineering (TE) extensions to OSPF version 2," RFC 3630, Oct. 2003.

[24] T. Li and H. Smit, "IS-IS extensions for traffic engineering," RFC 5305, Oct. 2008.

[25] L. Zhang, R. T. Braden, A. Terzis, and S. Vincent, "RSVP diagnostic messages," RFC 2745, Jan. 2000.

[26] B. Stephens, A. L. Cox, and S. Rixner, "Scalable multi-failure fast failover via forwarding table compression," in *Proc. Symposium on SDN Research (SOSR '16)*. ACM, 2016, article 9, pp. 1–12.

[27] J. Vanerio, S. Schmid, M. K. Schou, and J. Srba, "MPLS-Kit: An MPLS data plane toolkit," in *IEEE 11th International Conference on Cloud Networking (CloudNet)*, Nov. 2022, pp. 49–54.

[28] L. Andersson, I. Minei, and B. Thomas, "LDP specification," RFC 5036, Oct. 2007.

[29] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[30] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid, "Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst," in *Proc. ACM CoNEXT*, 2019, pp. 1–14.

[31] EthernetAlliance.org, "Ethernet jumbo frames," Nov. 2009. [Online]. Available: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf

[32] M. Chiesa, I. Nikolaevskiy, S. Mitrović, A. Gurtov, A. Madry, M. Schapira, and S. Shenker, "On the resiliency of static forwarding tables," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1133–1146, 2016.

[33] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. USENIX NSDI*, 2015, pp. 469–483.

[34] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM*, 2016, pp. 300–313.

[35] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. ACM SIGCOMM*, 2017, pp. 155–168.

[36] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *Proc. USENIX NSDI*, 2020, pp. 953–967.

[37] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *Proc. USENIX NSDI*, 2020, pp. 201–219.

[38] S. Bryant, C. Filsfils, S. Previdi, M. Shand, and N. So, "Remote loop-free alternate (LFA) fast reroute (FRR)," RFC 7490, Apr. 2015.

[39] S. Litkowski, P. Francois, A. Bashandy, C. Filsfils, and B. Decraene, "RFC draft: Topology independent fast reroute using segment routing," Tech. Rep., 2018. [Online]. Available: https://tools.ietf.org/html/draft-bashandy-rtgwg-segment-routing-ti-lfa-02

[40] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, "R3: Resilient routing reconfiguration," in *Proc. ACM SIGCOMM*, 2010, pp. 291–302.

[41] J. S. Arora, *Introduction to Optimum Design*, 4th ed. Boston: Academic Press, 2017.

[42] D. Applegate and E. Cohen, "Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs," in *Proc. ACM SIGCOMM*, 2003, pp. 313–324.

[43] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, "COPE: Traffic engineering in dynamic networks," in *Proc. ACM SIGCOMM*, 2006, pp. 99–110.

[44] O. Lemeshko, A. Romanyuk, and H. Kozlova, "Design schemes for MPLS fast reroute," in *2013 12th International Conference on the Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*, 2013, pp. 202–203.

[45] O. Lemeshko and K. Arous, "Fast ReRoute model for different backup schemes in MPLS-network," in *2014 First International Scientific-Practical Conference Problems of Infocommunications Science and Technology*, 2014, pp. 39–41.

[46] O. S. Yeremenko, O. V. Lemeshko, and N. Tariki, "Fast ReRoute scalable solution with protection schemes of network elements," in *2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON)*, 2017, pp. 783–788.

[47] O. Lemeshko and O. Yeremenko, "Linear optimization model of MPLS traffic engineering Fast ReRoute for link, node, and bandwidth protection," in *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*, 2018, pp. 1009–1013.

[48] J. Feigenbaum, B. Godfrey, A. Panda, M. Schapira, S. Shenker, and A. Singla, "Brief announcement: On the resilience of routing tables," in *Proc. Principles of Distributed Computing (PODC '12)*. ACM, 2012, pp. 237–238.

[49] S. Bryant, S. Previdi, and M. Shand, "A framework for IP and MPLS fast reroute using not-via addresses," RFC 6981, Aug. 2013.

[50] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica, "Achieving convergence-free routing using failure-carrying packets," in *Proc. ACM SIGCOMM*, 2007, pp. 241–252.

[51] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating MPLS networks," in *Proc. IEEE INFOCOM*, 2018, pp. 1799–1807.

A Proofs for Section 3.6

Theorem 1. Let N' be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. For all possible failure scenarios $F \subseteq E$, for all $e, e' \in E$ and $h, h' \in L^*$:

(1) if $\mathit{can\text{-}reach}^F_N(e, h, e', h')$ then $\mathit{can\text{-}reach}^F_{N'}(e, h, e', h')$,

(2) if $\mathit{must\text{-}reach}^F_N(e, h, e', h')$ then $\mathit{must\text{-}reach}^F_{N'}(e, h, e', h')$.

Proof. Before we prove (1) and (2), we first consider for any failure scenario F, the active forwarding tables τ_F and τ'_F for N and N', respectively. We argue by considering Line 17 and 21 in Algorithm 2 together with the definition of M that for any $v, \ell \in V \times L$,

(a) if $\tau_F(v, \ell) \neq \tau'_F(v, \ell)$ then $\tau_F(v, \ell) = \emptyset$,

since all new rules in τ' are appended with lower priority.

To prove (1), we assume that $\mathit{can\text{-}reach}^F_N(e, h, e', h')$ is true for some $e, e' \in E$ and $h, h' \in L^*$. Then there must exist a trace $(e_1, h_1) \dots (e_n, h_n)$ in N under F s.t. $(e_1, h_1) = (e, h)$ and $(e_n, h_n) = (e', h')$. For each step i, $1 \leq i < n$, we must have $\tau_F(tgt(e_i), head(h_i)) \neq \emptyset$, and hence due to (a) we have $\tau_F(tgt(e_i), head(h_i)) = \tau'_F(tgt(e_i), head(h_i))$. This means that the same trace is valid in N' under F, so $\mathit{can\text{-}reach}^F_{N'}(e, h, e', h')$ is also true.

To prove (2), we assume *must-reach*$^F_N(e, h, e', h')$ is true for some $e, e' \in E$ and $h, h' \in L^*$. Now all maximum traces in N under F starting from (e, h) contain (e', h'). Assume (to reach a contradiction) that some maximum trace $(e_1, h_1)(e_2, h_2)\ldots$ in N' under F with $(e_1, h_1) = (e, h)$ does not contain (e', h'). Then for some step i, (e_i, h_i) is contained in some maximum trace in N, while (e_{i+1}, h_{i+1}) is not. Hence, $\tau_F(tgt(e_i), head(h_i)) \neq \tau'_F(tgt(e_i), head(h_i))$, so due to (a) $\tau_F(tgt(e_i), head(h_i)) = \emptyset$. But then $(e_1, h_1)\ldots(e_i, h_i)$ is a maximum trace in N that does not contain (e', h'), which is a contradiction.

Theorem 2. Let $N' = (V, E', src', tgt', L', \tau')$ be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. If for all failure scenarios $F \subseteq E$ the network N satisfies *no-loops*^F_N then for all failure scenarios $F' \subseteq E$ the protected network also satisfies *no-loops*$^{F'}_{N'}$.

Proof. Assume (to reach a contradiction) that there exists a failure scenario F' such that *no-loops*$^{F'}_{N'}$ does not hold. Then there must be some infinite trace in the protected network $(e_1, h_1)(e_2, h_2)\ldots$ and because there are only finitely many links and labels, the infinite trace must consist of finitely many repeating heads $(e, head(h))$. Consider the first repeating head, i.e. the smallest b such that there exists $a < b$ where $(e_a, head(h_a)) = (e_b, head(h_b))$. The sequence $loop = (e_a, h_a)\ldots(e_b, h_b)$ is the (first) forwarding loop, and we shall now argue that it cannot exist.

Let $H = \{head(h_a), \ldots, head(h_b)\}$ be the set of head labels in the loop. Consider two cases: a) H consists only of protection labels, i.e. $H \subseteq (L' \setminus L)$, and b) H contains labels from the original data plane, i.e. $H \cap L \neq \emptyset$. Note that these two cases cover all possibilities.

For case a), since all protection labels are fresh, the loop must be only traversing protection paths. A single protection path does not contain a loop (assured by Definition 6), so it must be due to recursive protection moving the trace from one protection path to another eventually making a loop. Formally, let $loop^e_i$ denote the ith link e_{a+i} in the *loop* sequence, let $loop^h_i$ denote the ith header h_{a+i} in the *loop* sequence, and let p_j denote the jth link in the protection path p, i.e. $p_j = e_j$ if $p = e_1 \ldots e_j \ldots e_n$. Let $\langle e, p\rangle = protects(head(h_a))$ be the protection on which the loop starts, so e is a failed link. Let $e_a \ldots p_j$ be the longest part of p that coincides with $loop^e_1 \ldots loop^e_i$ such that $p_{j+1} \neq loop^e_{i+1}$. Since $head(loop^h_{i+1})$ is also a protection label, this must indicate a failure on $e' = p_{j+1}$, so we must be using a new protection $\langle e', p'\rangle = protects(head(loop^h_{i+1}))$, and the loop trace continues with $p'_1 \ldots p'_{j'} = loop^e_{i+1} \ldots loop^e_{i+j'}$ such that $p'_{j'+1} \neq loop^e_{i+j'+1}$. Here either *loop* merges back into the protection path p at some point after p_j, or $e'' = p'_{j'+1}$ is the next failed link. In the former case we can just forget the fully traversed protection $\langle e', p'\rangle$ and only consider the last recursive protection of a link e' on p. Since each protection path does not contain a loop, and the recursive protections always merge downstream (en-

sured by Line 19), the looping trace must eventually move to a new protection that is not fully traversed. This goes on until we reach the first failed link e on a protection path, which will complete the loop. In the protection graph there must be edges $(\langle e, p\rangle, \langle e', p'\rangle)$ annotated $\alpha(\langle e, p\rangle, \langle e', p'\rangle) = \{p_1, \ldots, e_a, \ldots, p_j\}$, and $(\langle e', p'\rangle, \langle e'', p''\rangle)$ annotated $\alpha(\langle e', p'\rangle, \langle e'', p''\rangle) = \{p'_1, \ldots, p'_{j'}\}$, and so on until the edge $(\langle e^{(k)}, p^{(k)}\rangle, \langle e, p\rangle)$, which forms a cycle. Note that all links in the annotations of the edges are part of the trace and hence cannot be failed, so the cycle is a bad cycle. Since the call to FindBadProtectionPairs on Line 11 of Algorithm 2 returns a set of protection-pairs that break all bad cycles, and Line 19 removes protection based on this set, there must be some (e_i, h_i) in *loop* where the given recursive protection is not installed, and hence the loop cannot exist.

For case b), the loop includes some routing from the original network, and hence the protection paths are fully traversed, so we can iteratively remove protection paths and corresponding failures and find a loop in the original network. Line 16 and Definition 7 along with Lines 4–10 and Line 19 ensures that a protection is only used if a higher priority entry has a path to the target of the protection path in the network with no failures. If that higher priority entry is part of another protection path, we will inductively remove that, eventually removing all failures, or else the entry is part of the original forwarding; hence, the original network will have a loop in some failure scenario $F \subseteq F'$. This contradicts the assumption that *no-loops*$_N^F$ holds for all F.

B Elaboration on Section 4.1

B.1 Data Plane Generation

MPLS-Kit [27] implements an abstraction of the distributed MPLS control plane, in which each router has its own protocol processes, yet these can directly access the memory of each other when required. This abstracts away communications.

MPLS introduces the concept of Forward Equivalence Class (FEC), which stands for the set of packets that should be forwarded in the same fashion; sent through the same outgoing interface to the same next-hop and executing the same set of header operations. Essentially, each FEC is identified with a local label on each router. When a packet arrives to a router, the latter determines to which FEC the former belongs to, and forwards it accordingly.

In MPLS-Kit, the two main control protocols that create Label Switched Paths (LSPs) by introducing forwarding entries on the router's tables, are LDP and RSVP. Both are industry standard, fully distributed protocols. Each LSP is related to a single FEC. LDP associates FECs with IP protocol prefixes and propagates labels through the network in order for the other routers to build

their own LSPs to reach said prefixes. RSVP builds tunnels (and associates them with FECs) from a given starting node (the *headend*) towards a final node (the *tailend*) over a path allowing for fine-grained packet steering. The routers along said path locally allocate labels to represent the LSP.

Given a weighted network topology and parameters for the control protocols as inputs, the tool outputs the data plane that results from letting the control protocols converge. As MPLS-Kit implements functionalities commonly used on ISP networks, the resulting data plane is then a *realistically-looking* MPLS data plane.

B.2 Simulation

As MPLS is a transport network, each user data packet (also called user-generated traffic) that enters the MPLS domain should follow an LSP and eventually exit the network. No successfully delivered data packets may be generated or terminated *inside* the MPLS domain.

To model the connections to the outside, we add a special node θ to V, and we add links (with infinite weight) between θ and a subset of MPLS routers that have interfaces with the outside. Such routers are known as Label Edge Routers (LERs). The links from θ are used to model the possible incoming packets, and the links to θ model the points where packets can leave the MPLS network.

Algorithm 3 shows how we simulate a packet, given the link where the packet starts and the intended exit link. Line 3 considers the possible next link-header pairs given the current link and header. Line 4 reports a failure if this set is empty, i.e. there is no valid rule for the current header. When there are multiple options due to nondeterminism, Line 5 randomly picks one. A maximum number of iteration is used to determine if the packet entered a forwarding loop.

The start and final links in the calls to Algorithm 3 are determined from the protocols used to create LSPs and their FECs. For the purpose of this section, a FEC f defines a mapping from a subset of routers V_f to the corresponding local labels for that FEC: $f : V_f \to L$.

For RSVP, each tunnel corresponds to a single FEC f in this protocol, and is implemented with an LSP from v to v'. We define links e, e' s.t. $src(e) = \theta$, $tgt(e) = v$, $src(e') = v'$, and $tgt(e') = \theta$, and we define $\tau(v', f(v')) = \{e', \texttt{pop}\}$ and initial header $h = f(v)$, where $\tau(v, f(v))$ contains the forwarding entries for the first step of the LSP. This encodes the behavior of the tunnel at the border of the MPLS domain. Simulating the header h is here an abstraction over how the forwarding is implemented on a real router. For the simulation with Algorithm 3, we use initial packet (e, h) and final link e', and we run this simulation for each tunnel.

In LDP, for each LDP FEC f (corresponding to an IP destination prefix ip)

Algorithm 3 Simulation of a packet starting from e_s.

Input: Network $N = (V, E, src, tgt, L, \tau)$, failures $F \subseteq E$, start $(e_s, h_s) \in E \times L^*$, final link $e_f \in E$
Output: Exit code (in $\{\texttt{SUCCESS}, \texttt{FAILURE}, \texttt{LOOP}\}$)

1: $(e, h) \leftarrow (e_s, h_s)$, $n \leftarrow 0$ ▷ *Initial packet*
2: **while** $n < \texttt{MAX\_TTL}$ **do**
3: $nexts \leftarrow \{(e', h') \mid (e', \omega) \in \tau_F(tgt(e), head(h)), h' = \mathcal{H}(h, \omega)\}$ ▷ *Compute all next hops*
4: **if** $nexts = \emptyset$ **then return** FAILURE
5: Pick at random $(e', h') \in nexts$
6: **if** $e' = e_f$ and $h' = \varepsilon$ **then return** SUCCESS ▷ *Success if egress router is reached*
7: $(e, h) \leftarrow (e', h')$, $n \leftarrow n + 1$ ▷ *Update packet*
8: **return** LOOP

announced by router v', we define a link e' s.t. $src(e') = v'$, $tgt(e') = \theta$, and we define $\tau(v', f(v')) = \{e', \texttt{pop}\}$. Let $X \subseteq V$ be the set of label edge routers. Then for each such router $v \in X$ we define a link e_v s.t. $src(e_v) = \theta$ and $tgt(e_v) = v$, and we define initial header $h_v = f(v)$, where $\tau(v, f(v))$ contains the forwarding entries of FEC f for packets entering the MPLS network at v with destination ip. We run for each v a simulation with Algorithm 3 using initial packet (e_v, h_v) and final link e', and we run such simulations for each LDP FEC. In our simulation we use $X = V$, which implies that all MPLS routers are LERs, and results in the maximum number of possible LDP generated LSPs.

In a nutshell, our simulator initializes an MPLS packet with a valid header to be handled by routers in the topology. For RSVP tunnels, this means a packet with proper headers on each headend router. For LDP entries, the simulator just initializes a packet with the corresponding label on each router.

C Artifact Appendix

C.1 Abstract

This appendix describes software artifacts associated with this work; the python source code of R-MPLS implemented on top of the MPLS data plane generator and simulator MPLS-Kit [27] (accepted for Global Internet 2022), along with a topology dataset derived from the original topology-zoo [29] with an adapted JSON format. These artifacts come with scripts to reproduce the experiments described in the evaluation section and a Jupyter notebook to process the result files and produce the paper statistics, Table 1, and Figures 4 and 5. The scripts are written in Bash and automate the execution of the

MPLS-Kit [27] python code. Additionally, we include a dataset containing the results files we obtained from executing the artifact's scripts on our computation cluster. Finally, we provide instructions for executing the scripts and reproducing the results.

C.2 Artifact check-list (meta-information)

- **Algorithm:** The code provided implements Algorithms 1 and 2 from the RMPLS paper on the file "rmpls.py", leveraging the MPLS-Kit [27] code base.
- **Data set:** A topology dataset in JSON format derived from the topology-zoo [29] dataset is provided. For the reviewers' convenience, a dataset with the results from executing the scripts of this artifact is also provided. The dataset is approximately 3.2GB in size.
- **Run-time environment:** The artifact should run on any Linux machine with Python3 and the required libraries. This setting is recommended: Linux kernel version 5.4.0 or later, Python 3.10. It may also require root/sudo access to install python modules.
- **Hardware:** The scripts use only CPU, memory, and I/O access, so in principle, they can be run on any Linux machine without tuning. Some simulations may require up to 20 GB RAM. As a reference, our installation used a cluster of 16 computing nodes with 1TB RAM each and 1248 CPU cores in total.
- **Execution:** Handled by the Bash scripts provided.
- **Experiments:** After installing the artifact, the experiment workflow can be reproduced by executing the provided scripts in the following order:
  1. Run "create_confs.sh {light|full}": creates configuration files and failure scenarios.
  2. Run "run.sh {light|full}": uses the configuration files and the topology dataset to create MPLS data planes and run simulations on each data plane and failure scenario.
  3. Activate the python virtual environment (e.g., running "source .venv/bin/activate").
  4. Run "jupyter-notebook make_plots.ipynb" and follow the instructions to open it in a browser. The notebook loads result files and produces figures and tables. In the "Options" cell, specify the parameters as appropriate.
- **Output:** The script "create_confs.sh" creates configuration specifications to be used as inputs by the data plane generator under the folder "confs/{light|full}/<topology name>". Each configuration file instructs the generation of an MPLS data plane from a specific set of protocols, as described in the paper. It also creates files describing the different possible failure scenarios up to $k = 2$ ("light") or $k = 4$ ("full") under a subfolder called "failure_chunks" for each topology. An additional folder "confs/conext22artifact/" contains the dataset of configurations and failure scenarios computed for the "full" case on our cluster.

The script "run.sh" takes the output from the previous script and generates the MPLS data planes, which include a topology, and based on the protocols in use, the forwarding tables on each router and a set of valid traffic source and destination pairs. Immediately after the script executes the simulation (tracing) on each data plane in each of the failure scenarios. The results are stored in JSON files under the "results/{light|full}/<topology name>" directory. There will be one result file for each configuration, summarizing results across all failure scenarios, according to the following format:

```
{"preamble":
  {"benchmark": <topology name>,
   "prog_alias": <configuration name>,
   "program": "Comparison of MPLS
              and R-MPLS"
  },
"stats":
  {"<code>":   {"comms": 532.0,
              "entries": 835.0,
              "k": 0,
              "loops": 0,
              "optimal": 121,
              "success": 121,
              "total": 121},
  ...
}
```

Code uniquely identifies each failure scenario. The stats are, respectively: no. of control-plane communications among routers, no. of LFIB entries, no. of failed links, no. of packet traces ended in forwarding loops, no. of possible successful traces, no. of actual successful traces, number of attempted packet traces.

An additional folder "confs/conext22artifact/" contains the dataset of configurations and failure scenarios as computed for the "full" case on our cluster for the evaluator's convenience.

The Jupyter notebook "make_plots.ipynb" loads the JSON files summarizing the results and the topology dataset and produces the PDF files with the plots from Figures 4 and 5 (stored in the "plots/" folder). It will also generate Table 1 showing success rates (percentages) and other performance statistics mentioned in the article.

- **How much disk space required (approximately)?:** At least 9 GB.
- **How much time is needed to complete experiments (approximately)?:** On our cluster installation, running the whole batch of experiments ("full") took five days; running on a standard laptop may take weeks. A smaller set of experiments ("light"), also included for the evaluator's convenience, can be run, taking up to 30 mins to complete in our installation while using 200MB RAM.
- **Publicly available?:** Yes.
- **Code and data licenses:** GNU General Public License v3.0
- **Archived:** `https://doi.org/10.5281/zenodo.7191618`

C.3 Description

How to access

The artifact is publicly available on Zenodo `https://zenodo.org/record/7191618` The provided results dataset uses 3.2GB of space. Reproducing the results using the "full" option will create an additional 3.2 GB of data. The remaining datasets and packages are usually below 2 GB. The total would be approximately 9 GB.

Software dependencies

This artifact assumes execution with a Debian-based OS machine or similar, with recommended requirements: Linux kernel version 5.4.0 or later, Python 3.8.10, jupyter-notebook 6.0.3.

The following Python libraries are also required: matplotlib 3.3.4, NetworkX 2.5, numpy 1.17.4, PyYAML 5.3.1, jsonschema 3.2.0, pandas 1.3.3, ujson 5.5.0.

Data sets

A JSON-formatted version of the publicly available topology-zoo [29] dataset is provided with the artifact. Also, a dataset containing the JSON result files we got from our execution of the experiment workflow is included.

C.4 Installation

After downloading and decompressing the artifact, change to the main artifact folder and run:

```
./install-dependencies.sh
```

C.5 Experiment workflow

Described in Section C.2. Detailed instructions can also be found on the README file.

C.6 Evaluation and expected results

After successful installation, the evaluator can run the artifacts' scripts as described in Section C.2. Detailed instructions can be found in the README file.

There are three different evaluation options, already mentioned in Section C.2. First, the "full" option will replicate the whole set of experiments

and produce their results, although it can take a significant amount of time. Alternatively, the "conext2artifact" can be used directly on the final jupyter notebook to generate the tables and figures from the article using our provided results dataset. Finally, the "light" option allows the evaluator to run a smaller and faster set of experiments to validate the artifact: no use of Plinko4, no LDP, failure scenarios considering up to two simultaneous link failures, and just the first 10 topologies from the dataset by alphabetical order. The results of this option will naturally differ from the published results, although the main findings hold.

Once the instructions from the README file have been completed (with the "full" option), the user will have launched a set of experiments for each kind of data plane included in the paper (RSVP, RSVP+R-MPLS Link, RSVP+R-MPLS Node, RSVP+FRR, RSVP+Plinko2, RSVP+Plinko4, LDP, LDP+R-MPLS Link, LDP+R-MPLS Node). They will also have reproduced Figures 4 and 5, Table 1, and other performance statistics from the article using the included jupyter notebook. The produced files' location is described in item C.2.

C.7 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

Paper B

MPLS-Kit: An MPLS Data Plane Toolkit

Juan Vanerio, Stefan Schmid,
Morten Konggaard Schou, and Jiří Srba

The paper has been published in:
IEEE 11th International Conference on Cloud Networking (CloudNet '22),
pp. 49-54, IEEE, 2022.
https://doi.org/10.1109/CloudNet55617.2022.9978791

The layout has been revised.

Abstract

Networking research often requires a means to quickly generate different realistic networks for evaluating the practical relevance. This is especially the case for emerging fields related to the automated verification of network configurations ("what-if analysis") or to AI-driven network operations ("self-driving networks"). Unfortunately, the data of real world network deployments are often scarce. In particular, while the topologies of many real communication networks have been made available online, this data typically does not include the routers' forwarding tables, e.g., by Internet Service Providers (ISPs). This introduces a dilemma, as generating arbitrary forwarding rules for these topologies may not adequately mimic network behavior.

We present MPLS-Kit, a tool for the automated generation of realistic MPLS data planes. In particular, the tool supports an efficient generation of MPLS data planes following widely-deployed industry-standard control protocols on top of arbitrary network topologies. Notably, MPLS-Kit supports the instantiation of MPLS Fast Reroute and VPN services. It further supports packet-level simulations providing a rich set of statistics about the simulated data plane which can be used for numerous applications, like congestion, latency, and resilience analysis. The generated data planes can be further exported in standard exchange formats and analyzed by formal verification tools.

1 Introduction

Modern communication networks are often very complex, and hence, modeling and analyzing their behavior can be difficult. Given that communication networks have become a critical infrastructure of our digital society in general and ISP networks in particular, and their dependable operation is crucial, this is worrisome.

In order to identify performance bottlenecks or to try out new innovative protocols, and verify their practical relevance, researchers need a means to generate realistic networks which mimic real and complex behavior. For example, in order to evaluate emerging automated network verification and what-if tools such as AalWiNes [1] and DeepMPLS [2], a way to generate realistic data planes (DP) is required. The generation of network configurations is also particularly important for emerging AI-driven approaches to improve the dependability and performance of networks, e.g., [2] or [3].

However, the data of real world network deployments are often scarce [4]. In particular, while the topologies of existing real communication networks have been made available online, e.g., [5], this data does not include the routers' forwarding tables; the latter however are required to model data planes. Introducing arbitrary forwarding rules may seem like a reasonable workaround, but the resulting data plane may vastly differ from the one found on a real

network. Research based on such data planes may give results far from what can be observed in practice.

Even researchers who manage to obtain such data, e.g., through a collaboration with industry, typically only have access to one or two complete network configurations. They are likely also not allowed to share their data with other researchers. Furthermore, while many network protocols are based on open standards and RFCs, many implementations of these protocols are either not open-source or not fully featured.

This paper presents MPLS-Kit, a toolset that allows the generation of synthetic data planes for the popular Multiprotocol Label Switching (MPLS) [6] system, which is widely deployed by ISPs. Concretely, given a weighted network topology the tool directly computes the MPLS data plane that would have been obtained after running commonly deployed MPLS protocols as Label Distribution Protocol (LDP [7]) and Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE [8]) until convergence.

Tool	Type	Direct DP computation	MPLS	LDP	RSVP FRR	MPLS VPN	Open-source MPLS Code	Parameterized configuration
GNS3 [9]	Emulator		√	√	√	√		
ns-3 [10]	Simulator		Limited				√	
Mininet [11]	Emulator		√	Limited			√	
Batfish [12]	Configuration analyzer	√					N/A	
OMNeT++ [13]	Simulator		√	√			√	
MPLS-Kit	Generator and Simulator	√	√	√	√	√	√	√

Table 1: Related work feature comparison.

Our Contributions. Our main contribution is MPLS-Kit, a tool and library to quickly generate MPLS data planes. The tool provides researchers and operators with fine control over the configuration of each network element, from fine path-level to automatic creation of a given number of tunnels, supporting the evaluation and analysis of MPLS configurations and their behavior under various conditions.

It also enables the study of the impact of different design choices on the network performance. MPLS-Kit includes utility tools that provide easy-to-use, simple interfaces for evaluation and automation of the execution.

The library is easy to extend and re-usable, allowing for fast prototyping of new concepts. In particular, MPLS-Kit supports RSVP-TE with Fast

ReRoute protections and VPN services, features rarely supported by other tools and forwarding stacks. MPLS-Kit further supports forwarding simulations at packet-level and provides easy-to-use, simple interfaces for evaluation and automation of the execution.

Together with this publication, MPLS-Kit is released as open source software at `https://github.com/juartinv/mplskit`, along with further examples of library usage. It can also be used online at `https://demo.AalWiNes.cs.aau.dk`.

Novelty and Related Work. We are not aware of any open-source tool which allows generation of realistic MPLS data planes. MPLS-Kit complements many existing works such as [1–3, 14] which so far relied on ad-hoc methodologies to generate data planes.

The lack of network data and protocol implementations, e.g., control planes, is not solved by emulators. GNS3 [9] allows running closed-source MPLS implementations that may use non-standard features and which cannot be independently reproduced for usage on research evaluations. Tools like ns-3 [10] and Mininet [11] can run open-source networking stacks but these may not implement all the expected protocol features. Besides, emulation of an entire network may also be time-consuming, as it requires the protocols to run until convergence, eventually providing the data plane as a by-product of the emulation.

In fact, realistic control plane emulators and/or generators of data planes are scarce in general. Batfish [12] is capable of building an internal vendor-agnostic network model (including the forwarding table) from complete and correct vendor configurations. Some useful features for research purposes, like using parametrized configurations or stating that some features should be created randomly are outside the scope of Batfish. Also, Batfish does not support MPLS networks.

MPLS-Kit may also be compared with OMNeT++ [13], a C++-based general discrete event simulation environment. OMNeT++ is often used for communication networks simulations. Although it provides limited support for MPLS, some required functionality is missing, like support for VPN services and MPLS failure-protection mechanisms. OMNeT++ is efficient for detailed, full-scale, cross-layer simulations and not for data plane generation and prototyping.

See Table 1 for a concise feature comparison among these tools.

2 MPLS Network Operation

We model an MPLS networks as a graph composed of *routers* interconnected through bidirectional *links*, forming a *topology*. MPLS networks are designed to transport packets from an ingress to an egress router.

Ingress packets. Packets are inspected by the ingress router on the edge of the MPLS domain that finds their *Forwarding Equivalence Class (FEC)*. FECs are used to represent a network resource or group of resources, such as traffic engineering tunnels or virtual private networks (VPNs). All packets that request the same resource must be forwarded the same way. Each router hosts a (control-plane) table called Label Information Base (LIB) mapping each FEC to a unique local label. After identifying the packet's FEC, the router initializes the packet's MPLS label stack with the corresponding label from the LIB, and then further processes it like an internal MPLS packet.
Internal MPLS packets. Each router has a Label Forwarding Information Base (LFIB) table that registers the prioritized forwarding instructions for each top of stack label. These instructions describe the outgoing interface and the operations (pop, swap, or push) to be performed on the packet's label stack. If a match is found, the router uses the highest priority forwarding rules such that the outgoing interface is up. This behavior enables the implementation of failure protections. If no match or no acceptable forwarding rule is found, the packet is dropped.

In a real MPLS network, the routers compute their LFIBs after exchanging FEC and label information through *MPLS control plane protocols* specialized in different FEC types. Additionally, the routers gain information about the topology through a Link-State Interior Gateway Protocol (IGP), typically OSPF or IS-IS. After exchanging messages, each router populates its local Link-State Database (LSDB) and the Traffic Engineering Database (TEDB) for traffic engineering functionalities.

We use the term *flow* to generalize over the reachability requirements of different FEC types. The flow specifies an initial router and header along with the destination routers allowed by the FEC.

An MPLS data plane is a network topology including its routers and links, together with the LFIB of each router. The primary purpose of MPLS-Kit is to generate such a data plane and to provide simulation capabilities on top of it.

3 MPLS-Kit Overview

MPLS-Kit is a modular Python library supporting MPLS data plane generation and packet-level simulation, following closely the operations described above.

Its main strengths are being stand-alone and extensible, producing MPLS data planes based on controllable industry-standard protocols. In terms of features, MPLS-Kit provides:

- per-platform label space,
- direct computation of converged data planes,

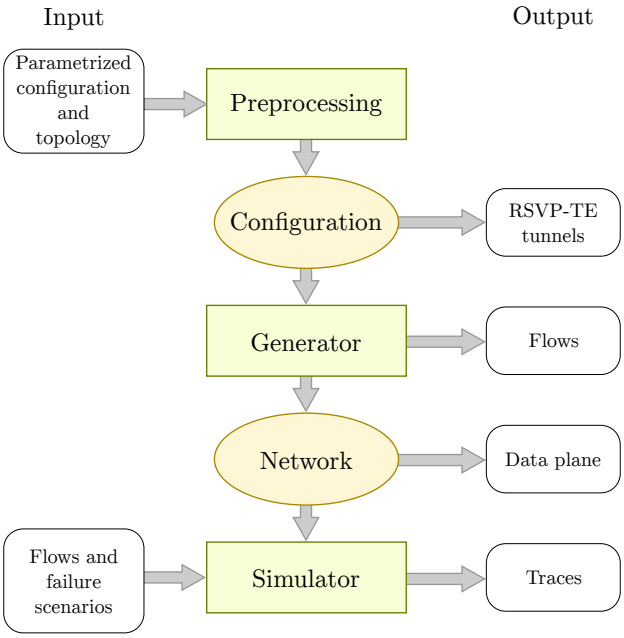


Fig. 1: MPLS-Kit interfaces and modules.

- support for Penultimate Hop Popping (PHP) ([15]),
- computation of Fast ReRoute (FRR) protection paths,
- instantiation of VPN services,
- deterministic and non-deterministic forwarding rules (supporting e.g., ECMP),
- multiple supported control-plane protocols,
- printing data planes and flows to files, and
- easy automation through command line interface and external configuration files.

The main overview of MPLS-Kit is depicted in Figure 1. Initially, a user provides a *parametrized configuration* including a topology (either a NetworkX [16] graph, an external file, or random generation instructions) and the set of enabled control plane protocols along with their parameters. For instance, to create traffic engineering tunnels, allowed values are a list of *(tunnel start, tunnel end)* tuples or a number of different tunnels to be randomly created. The parametrized configuration can be provided as YAML files, python variables, or command-line arguments.

The Preprocessing module receives a parametrized configuration and returns a concrete one, i.e., a configuration with no undetermined elements. On the tunnels example, a concrete configuration has an explicit topology (a NetworkX graph) and an explicit list of tuples specifying the requested tunnels.

As shown in Figure 1, the tool has two core modules; the Generator and the Simulator. The *Generator* is responsible for computing the data plane, i.e. the topology and the MPLS forwarding tables, according to the specification of the concrete configuration. Its operation reproduces the basic functionalities of the control plane protocols including keeping tables as the LIB.

The Generator returns a *Network* object including the data plane and control plane components. The list of valid flows in the network can be extracted by examining the LIBs and exporting them to a file. The data plane can be exported to a JSON file.

The *Simulator* module provides packet-level simulation functionality. Its inputs are a Network object, a list of flows to reproduce and a file describing failure scenarios. Each failure scenario consists of a list of failed links, such that forwarding instructions using them become unavailable. For each flow, the Simulator instantiates and forwards a packet while recording the trace of traversed links and the final result (e.g., succesful delivery at destination, detection of a forwarding loop, etc.). Results and traces can be exported to a file.

4 MPLS Dataplane Generation

MPLS-Kit uses high-level abstractions of the control plane components. Instead of thoughtfully mimicking control protocols and their subtleties, MPLS-Kit reproduces their essential functionalities in order to generate forwarding entries.

In a real MPLS network the routers engage in the distributed execution of an IGP protocol to obtain a consistent local view of the network topology to use in path computations. Such a process is time-consuming and prone to transient effects. Hence MPLS-Kit does not simulate any IGP, yet it does provide their essential features; i.e., providing a view of the topology and shortest path computations to every router, under the assumption of a single level, single area (OSPF or IS-IS) topology by default. This consideration covers most basic deployments, and the tool can be extended to multi-area deployments.

The communication involved in MPLS control protocols is also abstracted away; MPLS-Kit provides the protocol's client processes direct access to the memory content of their peer processes running on other routers. This simplification allows direct computation the same data plane that a real network achieves after convergence while avoiding delays and corner cases that arise

due to the interleaving of communication processes and protocol computations. The following protocols are implemented in MPLS-Kit.
Label Distribution Protocol (LDP) [7]. Provides connectivity when traffic engineering is not required. It works by establishing Label Switched Paths (LSPs) along the existing IP paths. The routers broadcast label mappings for each IP prefix to all of their neighbors. In turn, these neighbors allocate a local label to the prefix and broadcast the information further. MPLS-Kit assumes the implicit existence of an IP prefix for each link and node in the topology. LDP is a "best effort" protocol; if a router fails, all LSPs through it will fail.
Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE) [8]. Used between ingress and egress routers to establish tunnels implemented as LSPs, with desirable traffic engineering properties. Examples include waypointing, ensuring a given bandwidth and avoiding some network links. MPLS-Kit supports the *facility backup* protection method standardized on MPLS for protection of traffic engineering tunnels [17]. Here a backup LSP is established to protect a set of primary LSPs sharing a path segment by intersecting at the closest possible common downstream node [18].
VPN. MPLS-Kit also provides an MPLS client for instantiating a generalization of industry-standard MPLS VPN services such as Pseudo Wires [19], VPLS [20, 21] and VPRNs [22].

A high-level view of MPLS-Kit's generator internal architecture and its components is shown in Figure 2. Descriptions are provided in the following sections. Protocol-related parameters are adjustable.
Router. Supports multiple concurrent MPLS client processes implementing control plane functionalities. As each router object has direct access to the network topology (in the same way a real router has access to its local LSDB or TEDB), it also provides path computation functionalities.

It keeps the following local tables:

- Label Information Base (LIB): allocates a local label to each FEC. Each FEC is managed by a single MPLS client process on each router.
- Label Forwarding Information Base (LFIB): keeps routing entries for each local label. The routing entries are computed by the respective MPLS client.

Network. A network object is composed by a given topology, pointers to the routers and global functions. In MPLS-Kit, a topology is implemented as an undirected weighted graph whose nodes are routers, and its edges are links connecting the routers.
MPLS Client Process. Represents an actual process running on a router to participate in the MPLS control plane. There is a specialized client type per protocol, each responsible for:

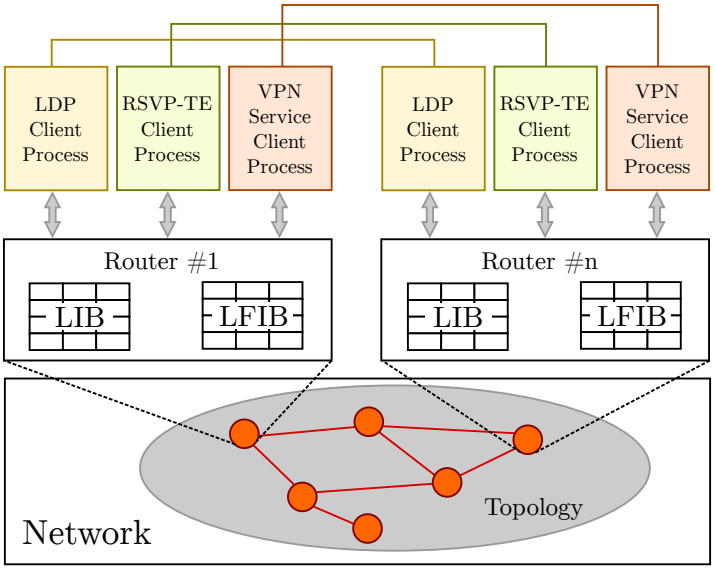


Fig. 2: Internal structure of the MPLS data plane generator.

- creating FEC objects for the network resources related to its control plane protocol (e.g., IP prefixes for LDP and TE tunnels for RSVP),
- requesting labels for its FECs on the router's LIB table, and
- providing functions to compute appropriate routing entries for LFIB building.

Performance Examples. We use MPLS-Kit to generate 100 data planes (one per topology) in 14.16s on an Ubuntu 20.04 system with Intel Core i9 and 32GiB RAM. The selected topologies are taken alphabetically from the Topology Zoo [5], accounting for 36.16 nodes 45.34 edges on average. and ranging between 5 and 197 nodes.

The following parameter values are used:

- PHP enabled,
- LDP enabled,
- RSVP-TE enabled; 20 random tunnels with facility protection FRR, and
- VPN services enabled; 15 instances spanning 5 nodes each.

Additionally, we generate data planes for random topologies of different sizes, using the same configuration as above, except with $3n$ RSVP-TE tunnels

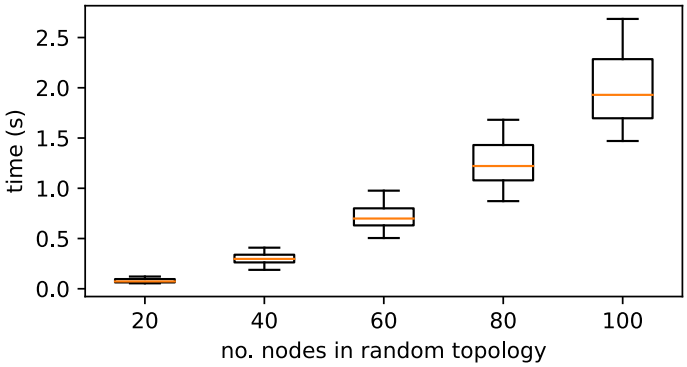


Fig. 3: Computation times for generating data planes.

Algorithm 1 Use of simulation to estimate link utilization.

Input: Flows $f \in F$, demands d_f, iterations n, capacity c_ℓ of each link ℓ.
Output: Utilization u_ℓ of each link ℓ

for each flow $f \in F$ **do**
  for i from 1 to n **do**
    `trace` := `Simulator.run`(f)
    for each link ℓ **do** $\texttt{use}_i(f, \ell)$:= count ℓ in `trace`
  for each link ℓ **do** $\texttt{avg\_use}(f, \ell) := \sum_{i=1}^{n} \texttt{use}_i(f, \ell)/n$
return $u_\ell := \sum_{f \in F} d_f \cdot \texttt{avg\_use}(f, \ell)/c_\ell$ for all links ℓ

and $2n$ VPN services with n being the number of nodes in the topology. For each n we generate 100 different topologies, one data plane per topology.

The results are shown in Figure 3. We can see that in the order of seconds, MPLS-Kit is capable of providing data planes fast enough for most interesting applications. These results are in line with MPLS-Kit's goal of providing a large variety of data to other tools in a time efficient way.

5 MPLS Forwarding Simulation

When a router forwards a packet in a real-world MPLS network, it uses the outmost label of the packet's stack to look up in its LFIB table. On a match, the router uses the forwarding rules with the highest priority such that the outgoing interface is up. If there are no acceptable forwarding rules at any priority level, or if at any point the time-to-live value reaches 0, the packet is dropped.

Consider the case in which the router's LFIB provides multiple equally

preferable forwarding rules for a packet. Solving such nondeterminism is actually outside the scope of MPLS, and on actual routers this decision is made by the lower-layer Forwarding Information Base. MPLS-Kit resolves it by choosing uniformly among all available options.

Packet Forwarding Simulation Implementation. In MPLS-Kit, at the beginning of each execution step a packet lies inside a router's memory. It is then processed and forwarded to the next hop (if any) moving to the next execution step, or finishing the simulation otherwise. As their real-world counterpart, MPLS-Kit's MPLS packets (instances of the MPLS packet class) have a label stack and a time-to-live field that decreases on each forwarding step. They also keep a pointer to the network object allowing access to all forwarding rules as well as to a list F of failed links. This information is used to filter out forwarding rules by priority when taking the forwarding decision.

The MPLS packet class implements the following forwarding methods:

- `step` simulates the next execution step as follows:
  1. In the current router's LFIB, identify the set of matching forwarding entries and their priorities.
  2. Filter out entries instructing to use a failed link, i.e. a link in F. If no entries remain, the packet is dropped.
  3. Select the highest priority rule. Break ties by randomly choosing with uniform distribution.
  4. Modify the packet's label stack, decrease its time-to-live, and send it to the next hop.
- `fwd` iteratively calls `step` until the packet depletes its label stack or its time-to-live expires.

As a packet is forwarded through the network, it keeps a record of its path (*traceroute*) for further analysis. Upon completion of the forwarding, the packet returns an exit code indicating its successful forwarding (0) or the specific type of error encountered.

Simulator Implementation. The Simulator class takes care of iteratively instantiating MPLS packets from user-specified flows on the network's routers with an adequate label stack and calling their `fwd` method.

Upon finalization, the Simulator gathers and aggregates statistics (exit code and traceroute) from all packet simulations and returns a summary of successful and failed cases. Results can be written to a CSV file or sent to the standard output.

Algorithm 2 Use of simulation for latency analysis.

Input: Flow f, delay $\texttt{delay}(\ell)$ for each link ℓ, iterations n
Output: Expected latency $E(\texttt{latency})$ for the flow f

```
for i from 1 to n do
    trace := Simulator.run(f)
    ℓ1ℓ2...ℓm := the sequence links in trace
    latency_i := Σ_{j=1}^{m} delay(ℓ_j)
return E(latency) := Σ_{i=1}^{n} latency_i / n
```

6 Use cases

We shall now provide examples of how MPLS-Kit can be used in practical applications.

Dataplane Verification. We can use the data plane output of MPLS-Kit to check properties using MPLS network data plane verification tools. The benefit of this use case is twofold. First, for testing and benchmarking a verifier tool, it is useful to have realistic data planes, since this is closer to what the tool will experience in real use. Second, the verifier allows us to check various properties of the generated data plane.

Formal verification. As a first example, we run AalWiNes [1] with a reachability query for each of the flows that the Generator outputs (see Figure 1). The query checks for a flow $(source, header, destinations)$ that a packet starting at the $source$ router with the given initial $header$ can reach one of the given $destinations$. For example for a flow (R_1, H, R_2), the query is $\langle H \rangle\ [\cdot \# R_1]\ \cdot^*\ [\cdot \# R_2]\ \langle \varepsilon \rangle$. This verifies that all the flows that MPLS-Kit claims to have created are in fact present in the output data plane.

Machine learning assisted verification. While formal verification tools for the data plane provide guaranteed results, they can still be relatively slow in practice. DeepMPLS [2] is a highly accurate, low execution time machine learning-based verification tool that also needs MPLS data planes as input, hence also benefitting from MPLS-Kit generation capabilities. The same properties (i.e., queries) verified with AalWiNes can be checked with DeepMPLS, and while the former provides guarantees, the latter can synthesize new MPLS header-rewriting rules in case a network property is not satisfied. Combined usage of AalWiNes, DeepMPLS and MPLS-Kit suggest an opportunity for synthesizing arbitrary property compliant MPLS data planes.

Congestion Analysis. We can use the simulation component of MPLS-Kit together with flow demands of a bandwidth-limited network to estimate congestion on links. From the trace output of the simulation, we can count how many times each flow traverses a certain link. By running the simulation of each flow multiple times, we can average out the randomness introduced by

Algorithm 3 Use of simulation for resilience analysis.

Input: Flows F, links L, failure-bound k, probability of single link failure p.
Output: Average success rate, weighted by failure probability

let $S := \{X \subseteq L \mid |X| \leq k\}$ ▷ *Failure scenarios*
for each X in S **do**
  $r_X := \texttt{Simulator.run}(F, X).\texttt{success\_rate}()$
  $w_X := p^{|X|} \cdot (1-p)^{k-|X|}$
return $\sum_{X \in S} r_X \cdot w_X / \sum_{X \in S} w_X$ ▷ *Normalized average*

multiple path. Now, given traffic demands for each flow and the capacity of each link, we can compute the link utilization as in Algorithm 1. All links ℓ with $u_\ell > 1$ can be congested given the traffic demands of the flows. The benefit of MPLS-Kit for this use case is the realistic packet-level simulation on an MPLS data plane that can be used e.g. in early stages of network design.

Latency Analysis. The trace output of the simulation in MPLS-Kit can be used to estimate latency of flows in the MPLS network. Measuring network latency in a simulated environment like the one provided by MPLS-Kit can be helpful in networks where the researcher or the operator has no access to trace collecting tools or the ability to inject test traffic into the network.

We can turn the trace into a sequence of links $\ell_1 \ell_2 \ldots \ell_m$, where m is the hop count. Due to nondeterminism in the data plane, several packets of the same flow may traverse different paths, so we repeat many experiments for the given flow. By labeling each link $\ell \in L$ in the topology with a delay, we can estimate the path latency of the flow as in Algorithm 2.

Resiliency Analysis. To test the resilience of various data planes, we can use the simulator's capability of simulating link failures. For a given set of failing links, the simulator outputs for each flow, whether the packet is successfully forwarded to an intended destination. We can run multiple simulations and compute the average success rate of packets in each failure scenario.

Using the external script, `create_confs.py`, we can for a data plane with links L systematically generate all k-failure scenarios as all the subsets $X \subseteq L$ with size $|X| \leq k$. To speed up the simulation, MPLS-Kit supports batch processing of failure scenarios and allows for possible parallelization of this computationally heavy task.

We can now average the success rates over all k-failure scenarios. To take failure probabilities into account, a weighted average can be used, as in the example in Algorithm 3, where link failures are modelled with a uniform, independent probability p. This gives a measure of the resilience of the given data plane. We can use this resiliency measure to compare different data planes of the same topology, for instance with or without fast re-route (FRR) protection.

7 Conclusions

Motivated by the need to produce realistic data planes for networking research, we developed MPLS-Kit, a library with MPLS data plane generation capabilities (including its fast-rerouting features) which also supports packet-level simulations. MPLS-Kit is designed to faithfully mimic the control plane processes responsible for computing the forwarding tables in real networks, especially ISP networks, without engaging in time-consuming full convergence simulations. Our design and implementation are based on a hierarchical structure of classes closely following the internal architecture of a router, allowing easy development of new functionalities and prototyping. As demonstrated with our use cases, MPLS-Kit can be useful in many scenarios, ranging from verification of data plane properties with external tools to simulation-based studies like convergence, latency and resiliency analysis. In summary, our contribution provides an opportunity to alleviate the scarceness of data plane datasets, thus encouraging more reproducible research in networking.

As a future work, we plan to extend our current work with Segment Routing (SR) [23], addressing first MPLS-SR and afterward moving towards IPv6-SR.

Acknowledgement

This research is supported by the Vienna Science and Technology Fund (WWTF) project ICT19-045 and by the DFF project QASNET.

References

[1] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[2] F. Geyer and S. Schmid, "DeepMPLS: Fast analysis of MPLS configurations using deep learning," in *Proc. IFIP Networking*, 2019.

[3] A. Blenk, P. Kalmbach, S. Schmid, and W. Kellerer, "O'Zapft is: Tap your network algorithm's big data!" in *Proc. ACM SIGCOMM 2017 Big-DAMA Workshop*, 2017, pp. 19–24.

[4] K. Claffy and D. Clark, "Comments on request for information on the american research environment," National Science and Technology Council (NSTC), 2020-01. [Online]. Available: https://catalog.caida.org/paper/2020_comments_rfi_american_research

[5] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[6] B. S. Davie and Y. Rekhter, *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.

[7] L. Andersson, I. Minei, and B. Thomas, "LDP specification," RFC 5036, Oct. 2007.

[8] D. O. Awduche, L. Berger, D.-H. Gan, T. Li, D. V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP tunnels," RFC 3209, Dec. 2001.

[9] "GNS3," https://gns3.com/, accessed: 2022-02-10.

[10] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[11] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proc. ACM SIGCOMM HotNets*, 2010, article 19, pp. 1–6.

[12] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. USENIX NSDI*, 2015, pp. 469–483.

[13] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. of ICST SIMUTools Workshop*, 2008.

[14] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[15] E. C. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta, "MPLS label stack encoding," RFC 3032, Jan. 2001.

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. 7th Python in Science Conference (SciPy 2008)*, 2008, pp. 11–15.

[17] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," RFC 4090, May 2005.

[18] M. Chiesa, A. Kamisinski, J. Rak, G. Retvari, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys and Tutorials (COMST)*, pp. 1253–1301, 2021.

[19] S. Bryant and P. Pate, "Pseudo wire emulation edge-to-edge (PWE3) architecture," RFC 3985, Mar. 2005.

[20] K. Kompella and Y. Rekhter, "Virtual private LAN service (VPLS) using BGP for auto-discovery and signaling," RFC 4761, Jan. 2007.

[21] M. Lasserre and V. Kompella, "Virtual private LAN service (VPLS) using label distribution protocol (LDP) signaling," RFC 4762, Jan. 2007.

[22] E. C. Rosen and Y. Rekhter, "BGP/MPLS IP virtual private networks (VPNs)," RFC 4364, Feb. 2006.

[23] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment routing architecture," RFC 8402, Jul. 2018.

Paper C

Faster Pushdown Reachability Analysis with Applications in Network Verification

Peter Gjøl Jensen, Stefan Schmid, Morten Konggaard Schou,
Jiří Srba, Juan Vanerio, and Ingo van Duijn

The paper has been published in:
Automated Technology for Verification and Analysis (ATVA 2021),
Lecture Notes in Computer Science, vol. 12971, pp. 170-186, Springer, 2021.
https://doi.org/10.1007/978-3-030-88885-5_12

The layout has been revised.

Abstract

Reachability analysis of pushdown systems is a fundamental problem in model checking that comes with a wide range of applications. We study performance improvements of pushdown reachability analysis and as a case study, we consider the verification of the policy-compliance of MPLS (Multiprotocol Label Switching) networks, an application domain that has recently received much attention. Our main contribution are three techniques that allow us to speed up the state-of-the-art pushdown reachability tools by an order of magnitude. These techniques include the combination of classic pre^* and $post^*$ saturation algorithms into a dual-search algorithm, an on-the-fly technique for detecting the possibility of early termination, as well as a counter-example guided abstraction refinement technique that improves the performance in particular for the negative instances where the early termination technique is not applicable. As a second contribution, we describe an improved translation of MPLS networks to pushdown systems and demonstrate on an extensive set of benchmarks of real internet wide-area networks the efficiency of our approach.

1 Introduction

Pushdown systems are a widely-used formalism with applications in, e.g., interprocedural control-flow analysis of recursive programs [1, 2] and model checking [3–6]. Pushdown systems have recently also received attention in the context of communication networks. Modern communication networks rely on increasingly complex router configurations which are difficult to manage by human administrators. Indeed, over the last years, several major network outages were due to human errors [7–10], and researchers are hence developing more automated and formal approaches to ensure policy compliance in networks. In particular, pushdown systems have been shown to enable fast automated what-if analysis of the policy compliance of an important and widely-deployed type of network, namely Multiprotocol Label Switching (MPLS) networks [11].

We are motivated by the objective to improve the performance of reachability analysis in pushdown systems, which typically relies on automata-theoretic approach for computing the pre^* and $post^*$ of a regular set of pushdown configurations [12]. Time is the most critical performance aspect of reachability analysis in general, and in particular, in the context of the increasingly large communication networks that need to be frequently reconfigured.

Our Contributions. We show that there is a significant potential to improve the state-of-the-art in reachability analysis of pushdown systems. In particular, we propose a fast on-the-fly early termination technique as well as an

algorithm that provides a novel combination of the classic pre^* and $post^*$ algorithms in order to harvest the benefits of both methods. We also suggest a specialization of the counter-example guided abstraction refinement (CEGAR) [13] technique that leverages equivalence classes on stack symbols as well as control states in order to improve the reachability analysis of MPLS networks that contain significant redundancy in the IP prefixes and produce a large number of MPLS labels (modeled as stack symbols). All techniques are general and apply to arbitrary pushdown systems, and are hence of interest in a wide range of applications. Finally, we also suggest a novel encoding approach of an MPLS communication network into a pushdown system that not only renders the pushdown analysis faster but also simpler compared to the recent approaches [11, 14, 15]. We report on our C++ prototype implementation and our empirical evaluation showing that the techniques can reduce the runtime by almost an order of magnitude compared to the state-of-the-art tools AalWiNes [15] and Moped [4].

Background and Related Work. We are motivated by the application of pushdown systems in order to perform automated what-if analysis of communication networks. In a nutshell, we consider a communication network interconnecting a set of routers which forward packets. The forwarding behavior of each router is defined by its pre-installed routing table which consists of a set of forwarding rules. To provide a dependable service, the network needs to fulfill a number of properties, such as reachability or loop-freedom, even under link failures.

Schmid and Srba recently showed in [11] that policy compliance of the widely-deployed MPLS networks can be verified in polynomial time, when overapproximating the possible link failures. Their approach leverages the fact that routing in MPLS networks is based on *label stacks*: packets contain stacks of labels which can be pushed and popped, and routers forward packets based on the top-of-stack label. Accordingly, these networks can be modelled as pushdown systems. In [14], the tool P-Rex was presented which implements the approach from [11]. P-Rex is implemented in Python, relies on the Moped model checker, and allows to verify complex network queries on network topologies with 20-30 routers in a matter of hours. The AalWiNes tool [15] is a follow-up work that improves the performance by an order of magnitude compared to P-Rex and replaces Moped with a tailored reachability engine written in C++.

In this paper, we show how to improve the performance by another order of magnitude compared to AalWiNes, by using three novel reachability techniques, including an early termination algorithm, a combined dual computation of pre^* and $post^*$, and a CEGAR approach. The CEGAR [13] technique was investigated in the context of symbolic pushdown systems before by Esparza

et al. [16] who consider sequential (recursive) programs whose statements are given as binary decision diagrams (BDDs). However, the CEGAR application is not used to speed up the reachability analysis but to refine the abstractions of the programs. Moped [12] is a model checker for linear-time logic on pushdown systems and has been adapted to many use cases. For instance, jMoped [5] models java byte-code as symbolic pushdown systems allowing automated analysis and verification of invariant properties with Moped.

2 Preliminaries

A *Labelled Transition System (LTS)* is a triple $(S, \Sigma, \rightarrow)$ where S is the set of *states*, Σ is the set of *labels* and $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation*. If $(s, a, s') \in \rightarrow$ then we write $s \xrightarrow{a} s'$. We also write $s \rightarrow s'$ if there is an $a \in \Sigma$ such that $s \xrightarrow{a} s'$ and let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$. The relation $\rightarrow^*$ can be annotated by the sequence of labels $w \in \Sigma^*$ as follows: $s \xrightarrow{\epsilon}{}^* s$ for any $s \in S$ where ε is the empty word, and $s \xrightarrow{aw}{}^* s'$ for $a \in \Sigma$ and $w \in \Sigma^*$ if $s \xrightarrow{a} s''$ and $s'' \xrightarrow{w}{}^* s'$ for some $s'' \in S$.

Definition 1. A *Nondeterministic Finite Automaton (NFA)* is a tuple $\mathcal{N} = (Q, \Sigma, \rightarrow, I, F)$ where Q is a finite set of *states*, Σ is a finite *input alphabet*, $\rightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the *transition relation*, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *accepting states*.

An NFA $\mathcal{N}$ *accepts* a word $w \in \Sigma^*$ if the LTS $(Q, \Sigma, \rightarrow)$ satisfies $q_0 \xrightarrow{w}{}^* q_f$ for an initial state $q_0 \in I$ and an accepting state $q_f \in F$. The language $Lang(\mathcal{N})$ is the set of all words that $\mathcal{N}$ accepts.

Definition 2. A *Pushdown System (PDS)* is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of *control locations (states)*, Γ is a *stack alphabet*, and the set of *rules* Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$. If $((p, \gamma), (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$.

A *configuration* of a pushdown system is a pair $\langle p, w \rangle$ where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted $Conf(\mathcal{P})$. The semantics of a pushdown system $\mathcal{P}$ is given by the LTS $\mathcal{T}_{\mathcal{P}} = (Conf(\mathcal{P}), \Delta, \Rightarrow_{\mathcal{P}})$ where $\langle p, \gamma w' \rangle \overset{r}{\Rightarrow}_{\mathcal{P}} \langle p', ww' \rangle$ for all $w' \in \Gamma^*$ whenever there is $r = ((p, \gamma), (p', w)) \in \Delta$. If $\mathcal{P}$ is clear from the context, we may omit it from $\hookrightarrow_{\mathcal{P}}$ and $\Rightarrow_{\mathcal{P}}$. We only consider normalized PDS in which all rules $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ satisfy $|w| \leq 2$. Note that any PDS can be normalized by adding at most $\mathcal{O}(|P|)$ auxiliary states [12].

Definition 3. Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS. A *$\mathcal{P}$-automaton* is an NFA $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ with the stack symbols of $\mathcal{P}$ as its input alphabet and with the initial states being the control locations of $\mathcal{P}$.

A $\mathcal{P}$-automaton accepts a pushdown configuration $\langle p, w\rangle$ of $\mathcal{P}$ if $p \xrightarrow{w}{}^* q$ for some $q \in F$. The set of all configurations accepted by $\mathcal{A}$ is denoted by $Lang(\mathcal{A})$. A set of configurations is called *regular* if it is accepted by some $\mathcal{P}$-automaton.

Problem 1 (Pushdown Reachability Problem). For a PDS $\mathcal{P}$ and two regular sets of configurations C and C', is there $c \in C$ and $c' \in C'$ such that $c \overset{\sigma}{\Rightarrow}{}^*_{\mathcal{P}} c'$ for some sequence of rules σ? In the affirmative case return a witness trace (c, σ).

Given a PDS $\mathcal{P}$ and a set of configurations $C \subseteq Conf(\mathcal{P})$ the *predecessors* are defined as $pre^*(C) = \{c \mid \exists c' \in C, c \Rightarrow^* c'\}$ and the *successors* as $post^*(C) = \{c \mid \exists c' \in C, c' \Rightarrow^* c\}$. If C is a regular set of configurations, then both $pre^*(C)$ and $post^*(C)$ are also regular sets of configurations [17].

Construction of pre^*. Given a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$, we construct a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*} = (Q, \Gamma, \rightarrow, P, F)$ where $\rightarrow$ is obtained by repeatedly adding transitions to $\rightarrow_0$ according to the following saturation rule: if $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ and $p' \xrightarrow{w}{}^* q$ in the current automaton, add a transition $p \xrightarrow{\gamma} q$.

Theorem 1 ([6, 12, 18]). *An automaton $\mathcal{A}_{pre^*}$ that satisfies $Lang(\mathcal{A}_{pre^*}) = pre^*(Lang(\mathcal{A}))$ can be built in $\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time and $\mathcal{O}(|Q| \cdot |\Delta| + |\rightarrow_0|)$ space.*

There is a slightly more complicated saturation procedure for $\mathcal{A}_{post^*}$.

Theorem 2 ([6, 12, 18]). *An automaton $\mathcal{A}_{post^*}$ that satisfies $Lang(\mathcal{A}_{post^*}) = post^*(Lang(\mathcal{A}))$ can be built in $\mathcal{O}(|P| \cdot |\Delta| \cdot (n_1 + n_2) + |P| \cdot |\rightarrow_0|)$ time and space, where $n_1 = |Q \setminus P|$ and n_2 is the number of different pairs (p, γ) such that there is a rule of the form $\langle p', \gamma'\rangle \hookrightarrow \langle p, \gamma\gamma''\rangle$ in Δ.*

Problem 1 can now be solved in polynomial time using either the pre^* or $post^*$ algorithm by computing e.g. $pre^*(C')$ and checking if $C \cap pre^*(C') \neq \emptyset$, similarly for $post^*$, relying on the fact that regular languages are closed under intersection. A witness trace σ can be computed by storing metadata during the saturation procedures (see e.g. [12] for details).

3 Formal Model of MPLS Networks

An MPLS network consists of a topology and forwarding rules.

Definition 4. A *network topology* is a directed multigraph (V, E, s, t) where V is a set of *routers*, E is a set of *links* between routers, $s : E \rightarrow V$ assigns the *source router* to each link, and $t : E \rightarrow V$ assigns the *target router*.

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. A link is *active* if it belongs to $E \setminus F$.

For a nonempty set of MPLS labels L, we define the set of *MPLS operations* on packet headers as $Op(L) = \{\mathtt{swap}(\ell) \mid \ell \in L\} \cup \{\mathtt{push}(\ell) \mid \ell \in L\} \cup \{\mathtt{pop}\}$. We define the semantics of MPLS operations $[\cdot] : Op(L) \to (L \to L^*)$ by $[\mathtt{pop}](\ell) = \varepsilon$, $[\mathtt{swap}(\ell')](\ell) = \ell'$ and $[\mathtt{push}(\ell')](\ell) = \ell'\ell$ for all $\ell, \ell' \in L$.

The forwarding of a packet in an MPLS network depends on the interface (link) that the packet arrives on, which determines the forwarding table used, and the top MPLS label in the packet header, which is used for lookup in the forwarding table. When a packet enters the MPLS domain, it does not yet have any MPLS label, and the forwarding depends only on the link that it arrives on as well as the type of the protocol that is used for the packet forwarding (this is abstracted away by the use of nondeterminism).

Definition 5. An *MPLS network* is a tuple $N = (V, E, s, t, L, \tau)$ where (V, E, s, t) is a network topology, L is a finite set of MPLS labels, and $\tau : E \cup (E \times L) \to \left(2^{E \times Op(L)^+}\right)^*$ is the routing table.

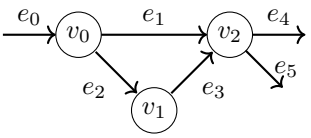
For every link $e \in E$ and for every link-label pair $(e, \ell) \in E \times L$, the routing table returns a sequence of *traffic engineering groups* $O_1 O_2 \ldots O_n$ where each group is a set of the form $\{(e_1, \omega_1), \ldots, (e_m, \omega_m)\}$ where e_j is the outgoing link such that $t(e) = s(e_j)$ and $\omega_j \in Op(L)^+$ is a nonempty sequence of MPLS operations to be performed on the packet header. Figure 1a gives an example of an MPLS network with its routing table in Figure 1b. Here the priority column refers to the index of the corresponding traffic engineering group.

The semantics of a traffic engineering group is that any pair of active link and operation sequence in the group can be nondeterministically chosen, hence abstracting away from various specific routing policies that allow e.g. splitting a flow along multiple paths. The group O_i has a higher priority than O_{i+1}, and during forwarding the router always selects the traffic engineering group with the highest priority and at least one active link.

For a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \ldots, (e_m, \omega_m)\}$ let $E(O) = \{e_1, e_2, \ldots, e_m\}$ denote the set of outgoing links in the group.

Definition 6. For a set of failed links $F \subseteq E$ we define the *active routing table* $\tau_F : E \cup (E \times L) \to 2^{E \times Op(L)^+}$ as $\tau_F(u) = \{(e', \omega) \in \mathcal{A}_F(\tau(u)) \mid e' \in E \setminus F\}$, where $u = e$ or $u = (e, \ell)$ and $\mathcal{A}_F$ is the *active traffic engineering group* defined as $\mathcal{A}_F(O_1 O_2 \ldots O_n) = O_j$ if j is the lowest index such that $E(O_j) \setminus F \neq \emptyset$, or $\mathcal{A}(O_1 O_2 \ldots O_n) = \emptyset$ if no such j exists.

Definition 7. The semantics of MPLS operations is a partial *header rewrite function* $\mathcal{H} : L^* \times Op(L)^* \rightharpoonup L^*$, where $\omega, \omega' \in Op(L)^*$, $h \in L^*$ and ε is the empty



(a) Network topology

Router	e_{in}	Label	Priority	e_{out}	Operation
v_0	e_0	–	1	e_1	push(11)
	e_0	–	2	e_2	push(11) ∘ push(20)
	e_0	10	1	e_1	swap(12)
	e_0	10	2	e_2	swap(12) ∘ push(20)
v_1	e_2	20	1	e_3	pop
v_2	e_1	11	1	e_4	pop
	e_1	12	1	e_5	pop
	e_3	11	1	e_4	pop
	e_3	12	1	e_5	pop

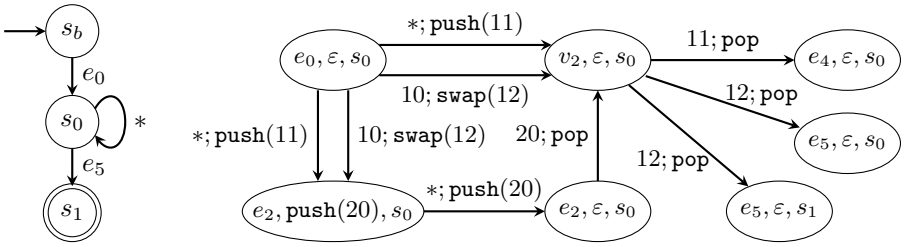
(b) Routing table

$\sigma_1 = (e_0, \varepsilon)(e_1, 11)(e_4, \varepsilon)$ for $F = \emptyset$

$\sigma_2 = (e_0, 10 \circ 30)(e_2, 20 \circ 12 \circ 30)(e_3, 12 \circ 30)(e_5, 30)$ for $F = \{e_1\}$

$\varphi = \langle 10 \cdot^* \rangle\ e_0\ \cdot^*\ e_5\ \langle 30 \rangle\ 1$ is satisfied by σ_2

(c) Example traces σ_1 and σ_2 under a set of failed links F, and an example query φ



(d) Corresponding pushdown system for the query φ. Left: the NFA $\mathcal{N}_b$ for φ. Right: the generated PDS $\mathcal{P}$. The labelled arrow $(p) \xrightarrow{\ell; op} (p')$ denotes the rule $\langle p, \ell \rangle \hookrightarrow \langle p', [op](\ell) \rangle$. The state (v_2, ε, s_0) is merged from (e_1, ε, s_0) and (e_3, ε, s_0).

$P_i = \{(e_0, \varepsilon, s_0)\}$

$P_f = \{(e_5, \varepsilon, s_1)\}$

$Lang(\mathcal{N}_i) = \{10 \circ w \circ \bot \mid w \in L^*\}$

$Lang(\mathcal{N}_f) = \{30 \circ \bot\}$

(e) Initial/final configurations for φ

$\langle (e_0, \varepsilon, s_0), 10 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (e_2, \texttt{push}(20), s_0), 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (e_2, \varepsilon, s_0), 20 \circ 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (v_2, \varepsilon, s_0), 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}} \langle (e_5, \varepsilon, s_1), 30 \circ \bot \rangle$

(f) Computation in PDS $\mathcal{P}$ corresponding to σ_2

Fig. 1: Example of a small network and its encoding into a pushdown system

sequence of operations:

$$\mathcal{H}(h, \omega) = \begin{cases} h & \text{if } \omega = \varepsilon \\ \mathcal{H}([op](\ell) \circ h', \omega') & \text{if } \omega = op \circ \omega' \text{ and } h = \ell \circ h' \text{ with } \ell \in L, h' \in L^* \\ \textit{undefined} & \textit{otherwise}\ . \end{cases}$$

Using the example from Figure 1, the operation sequence $\texttt{swap}(12) \circ \texttt{push}(20)$ applied to the header $10 \circ 30$ yields $\mathcal{H}(10 \circ 30, \texttt{swap}(12) \circ \texttt{push}(20)) = 20 \circ 12 \circ 30$.

Definition 8. A *trace* in a network $N = (V, E, s, t, L, \tau)$, given a set of failed links $F \subseteq E$, is any finite sequence $(e_1, h_1)(e_2, h_2) \dots (e_n, h_n) \in \big((E \setminus F) \times L^*\big)^*$ of link-header pairs where for all i, $1 \leq i < n$, $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \tau_F(u)$, where either $u = e_i$ or $u = (e_i, head(h_i))$, where $head(h_i)$ is the top (left-most) label of h_i. If $h_i = \varepsilon$ then $head(h_i)$ is undefined.

In Figure 1c we can see a trace σ_1 in the network without any failed links, while for the failure set $F = \{e_1\}$ we notice that σ_1 is not a trace, while σ_2 is.

3.1 MPLS Network Verification

Similar to prior work [14, 15], we present a powerful query language that allows us to specify regular trace properties, both regarding the initial and final label-stacks as well as the sequence of links in the trace.

Definition 9. A reachability *query* for an MPLS network $N = (V, E, s, t, L, \tau)$ is of the form $\langle a \rangle\ b\ \langle c \rangle\ k$ where a and c are regular expressions over the set of labels L, b is a regular expression over the set of links E, and $k \geq 0$ specifies the maximum number of failures to be considered.

We assume here a standard syntax for regular expressions and by $Lang(a)$, $Lang(b)$ and $Lang(c)$ we understand the regular language defined by the expressions a, b and c, respectively. Intuitively, the query $\langle a \rangle\ b\ \langle c \rangle\ k$ asks if there is a network trace such that the initial header (stack of labels) belongs to $Lang(a)$, the sequence of visited links belongs to $Lang(b)$ and at the end of the trace the final header belongs to $Lang(c)$.

We further use the following notation for specifying links in the network. If v and u are routers, then $[v\#u]$ matches any link e from v to u such that $s(e) = v$ and $t(e) = u$. The dot-syntax is used to denote any link or label in the network and it is extended to match also any router so that $[v\#\cdot] = \bigcup_{u \in V} [v\#u]$ and $[\cdot\#u] = \bigcup_{v \in V} [v\#u]$.

Problem 2 (Query Satisfiability Problem). Given an MPLS network N and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, decide if there exists a trace $\sigma = (e_1, h_1) \dots (e_n, h_n)$ in the network N for some set of failed links F such that $|F| \leq k$ where $h_1 \in Lang(a)$, $e_1 \dots e_n \in Lang(b)$, and $h_n \in Lang(c)$. If this is the case, the query φ is *satisfied* and we call σ a *witness trace*.

In Figure 1c the query φ asks if a packet with the top most label 10 can be forwarded from the link e_0 to e_5, while just leaving the label 30 on the label-stack. This query is satisfied and the trace σ_2 serves as a witness trace. On the other hand, the query $\langle\ \cdot^*\rangle\ [\cdot\#v_1]\ \cdot^*\ e_3\ \langle\ \cdot^*\rangle\ 0$ is not satisfied as it asks if a packet (with any header) arriving on some link to the router v_1 (note that e_0 is the only such link) can reach the link e_3 if no links fail. Such a trace exists only if we allow for at least one failed link.

3.2 From Query Satisfiability to Pushdown Reachability

We now solve the query satisfiability problem by translation to the pushdown reachability problem. This is an over-approximation, so in a few cases a positive result cannot be transfered back to the query satisfiability problem. Notice that our construction is different from the one in [14]. In particular, we model the initial and final headers directly as NFA rather than simulating them with PDSs, which makes the reduction simpler and more efficient at the same time.

The behavior of the network for a fixed set of failed links F is given by the active routing table τ_F, however to represent the possible behavior for any set of failed links F with $|F| \leq k$, we use the following definition.

Definition 10. For a network $N = (V, E, s, t, L, \tau)$ and number k, we define the *overapproximating routing table* $\tau^k(u) = \bigcup_{j=1}^{i} O_j$, where $\tau(u) = O_1 O_2 \dots O_n$ and i is the smallest index such that $|\bigcup_{j=1}^{i} E(O_j)| > k$.

The routing table τ^k overapproximates all possible routing table entries if up to k links fail at any router.

Given a network $N = (V, E, s, t, L, \tau)$ and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, let $\mathcal{N}_a = (S_a, L, \to_a, \{s_a\}, F_a)$, $\mathcal{N}_b = (S_b, E, \to_b, \{s_b\}, F_b)$ and $\mathcal{N}_c = (S_c, L, \to_c, \{s_c\}, F_c)$ be the NFAs corresponding to the regular expressions a, b and c. Let $L_\perp = L \cup \{\perp\}$ where $\perp$ is used to represent the bottom of the stack. We construct a PDS $\mathcal{P} = (P, L_\perp, \Delta)$ where $P = E \times \overline{Ops} \times S_b$ and $\overline{Ops}$ is the set of all operation sequences and suffixes hereof occurring in τ^k. The set of rules Δ is defined by:

a) $\langle (e, \varepsilon, s), \ell \rangle \hookrightarrow \langle (e', \omega, s'), [op](\ell) \rangle$ if $s \xrightarrow{e'}_b{}^{*} s'$ and $(e', op \circ \omega) \in \tau^k(u)$ where either (i) $u = (e, \ell)$, or (ii) $u = e$, $\ell \in L$, or (iii) $u = e$, $\ell = \perp$, $op = \texttt{push}(\ell')$.

b) $\langle (e, op \circ \omega, s), \ell \rangle \hookrightarrow \langle (e, \omega, s), [op](\ell) \rangle$ for $\ell \in L$ and for $\ell = \perp$ if $op = \texttt{push}(\ell')$.

Finally, we define the initial states $P_i = \{(e, \varepsilon, s) \mid e \in E, s \in S_b, s_b \xrightarrow{e}_b{}^{*} s\}$, and the final states $P_f = \{(e, \varepsilon, s_f) \mid e \in E, s_f \in F_b\}$. Let $\mathcal{N}_\perp$ be an NFA such that $Lang(\mathcal{N}_\perp) = \{\perp\}$. Let $\mathcal{N}_i = \mathcal{N}_a \circ \mathcal{N}_\perp$ and $\mathcal{N}_f = \mathcal{N}_c \circ \mathcal{N}_\perp$ where $\circ$ is the standard NFA concatenation operator. For the running example this is shown in Figure 1e. Now the query satisfiability problem is reduced to the problem of finding configurations $c \in P_i \times Lang(\mathcal{N}_i)$ and $c' \in P_f \times Lang(\mathcal{N}_f)$ such that $c \overset{\sigma}{\Rightarrow}_{\mathcal{P}}^{*} c'$, and in the positive case outputing the trace (c, σ).

Optimizations. To reduce the size of the PDS we use the following optimizations. We merge control locations (e, ω, s) and (e', ω, s) for which $t(e) = t(e')$, $\tau(e) = \tau(e')$ and $\tau(e, \ell) = \tau(e', \ell)$ for all $\ell \in L$, i.e. the lookup is independent of which interface on the router the packet arrives on, which is often the case

in many existing networks. We only construct control states that are reachable from P_i. If a rule $\langle p, \ell \rangle \hookrightarrow \langle p', [op](\ell) \rangle$ is added for all $\ell \in L_\perp$, we represent it succinctly as $\langle p, * \rangle \hookrightarrow \langle p', [op](*) \rangle$ where $*$ is a wildcard representing any label. The wildcard can be handled efficiently by our $post^*$ algorithm, while for pre^* it needs to be unfolded. In Figure 1d we can see the generated pushdown system for our running example and in Figure 1f we show an execution of the pushdown system corresponding to the network trace σ_2.

We can now show that if there is a network trace satisfying a given query then the constructed pushdown system provides a positive answer in the reachability analysis.

Theorem 3. *Given a network N and a query φ, if there exists a witness trace in the network satisfying φ, then there exist $c \in P_i \times Lang(\mathcal{N}_i)$, $c' \in P_f \times Lang(\mathcal{N}_f)$ and $\sigma \in \Delta^*$ such that $c \overset{\sigma}{\Rightarrow}^*_{\mathcal{P}} c'$.*

Proof (Sketch). By induction on the length of the witness trace we construct the corresponding pushdown execution following the construction of the pushdown rules Δ. One step in the network trace can be simulated by a sequence of pushdown transitions as the rules of type b) apply the MPLS operations sequentially one by one. □

For the other direction, we have to first make sure that the trace obtained from the execution in the pushdown system is indeed a valid network trace (since the pushdown system overapproximates the set of all valid traces as it assumes that at any router, up to k links can fail).

Reconstruction of Network Traces. The reachability analysis for the pushdown system $\mathcal{P}$ returns (in the affirmative case) a trace $\langle p_0, w_0 \rangle \overset{r_1}{\Rightarrow}_{\mathcal{P}} \dots \overset{r_m}{\Rightarrow}_{\mathcal{P}} \langle p_m, w_m \rangle$. We extract (e, h) for every i such that $p_i = (e, \omega, s)$ and $w_i = h \circ \perp$ where $\omega = \varepsilon$, producing a network trace $(e_0, h_0) \dots (e_n, h_n)$. For each rule r of type a) that was added due to $(e', \omega) \in \tau^k(u)$, we define $F^\tau(r) = \bigcup_{j=1}^{i-1} E(O_j)$ where $\tau(u) = O_1 O_2 \dots$ and i is the smallest index such that $(e', \omega) \in O_i$. Let $F = \bigcup_{i=1}^{n} F^\tau(r_i)$ be the set of failed links in order to enable the execution of the trace. Now we have to check that $\{e_0, \dots, e_n\} \cap F = \emptyset$ and $|F| \leq k$ in order to guarantee that the corresponding network trace is executable; otherwise the overapproximation returns an inconclusive answer.

Theorem 4. *Given a network N and a query φ, if in the constructed pushdown system there exist $c \in P_i \times Lang(\mathcal{N}_i)$, $c' \in P_f \times Lang(\mathcal{N}_f)$ and $\sigma \in \Delta^*$ s.t. $c \overset{\sigma}{\Rightarrow}^*_{\mathcal{P}} c'$ from which a valid network trace σ' can be reconstructed, then σ' satisfies φ.*

Proof (Sketch). From the construction of the pushdown system and the encoding of MPLS operations by a series of pushdown transitions, we can see that if the reconstructed trace only uses active links, i.e. $\{e_0, \dots, e_n\} \cap F = \emptyset$, then it

corresponds to a correct network trace for the routing table τ^k. However, as τ^k allows for up to k link failures at any router along the trace, the total number of failed links along the reconstructed trace may exceed the bound k. This is detected in the trace reconstruction procedure. □

4 Improving Pushdown System Reachability Analysis

We now describe our improvements to the pushdown reachability analysis.

4.1 Early Termination of Reachability Algorithms

In Section 2 we showed that for a given PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{P}$-automaton $\mathcal{A}$ that represents a set of configurations in $\mathcal{P}$, we can construct the $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ automata by iteratively adding additional transitions to the existing automaton $\mathcal{A}$. During this saturation procedure, the language of the current $\mathcal{P}$-automaton $\mathcal{A}$ can only increase (w.r.t. subset inclusion). Hence if at any point the current $\mathcal{P}$-automaton has a nonempty intersection with some set of target configurations, it will have the nonempty intersection also after the saturation procedure terminates. We can hence allow for an early termination as we can return a witness trace before completing the saturation procedure.

We further generalize this idea by considering $\mathcal{P}$-automata $\mathcal{A}_1 = (Q_1, \Gamma, \to_1, P, F_1)$ and $\mathcal{A}_2 = (Q_2, \Gamma, \to_2, P, F_2)$ that can be step-by-step saturated by calling (in arbitrary order) the functions AddTransition($q_1 \xrightarrow{\gamma}_1 q_1'$) and AddTransition($q_2 \xrightarrow{\gamma}_2 q_2'$), respectively. Each such call will add the corresponding transition in its argument to the automaton $\mathcal{A}_1$ resp. $\mathcal{A}_2$ and at the same time compute the reachable part (stored in the nondecreasing set R of pairs of states in $\mathcal{A}_1$ and $\mathcal{A}_2$) of the product automaton $\mathcal{A}_\cap$ representing the current intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$. The function call AddTransition($q_i \xrightarrow{\gamma}_i q_i'$) where $i \in \{1, 2\}$ relies on the function AddState(q_1, q_2) given in Algorithm 1 and before any calls to AddTransition are made, it is assumed that the product automaton is initialized by calling AddState(p, p) for all states $p \in P$. The algorithm exits (early terminates) and returns true as soon as the product automaton accepts at least one string.

Proposition 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two initial $\mathcal{P}$-automata and let $\mathcal{A}_1'$ and $\mathcal{A}_2'$ be the resulting $\mathcal{P}$-automata after an arbitrary number of calls to the function* AddTransition *given in Algorithm 1. Then $Lang(\mathcal{A}_\cap) = Lang(\mathcal{A}_1') \cap Lang(\mathcal{A}_2')$ and as soon as $Lang(\mathcal{A}_\cap) \neq \emptyset$, the algorithm returns true.*

This on-the-fly detection of nonemptiness of the intersection between two $\mathcal{P}$-automata can be used to allow for early termination when deciding the

Algorithm 1 On-the-fly computation of product automaton

Input: $\mathcal{P}$-automata $\mathcal{A}_1 = (Q_1, \Gamma, \rightarrow_1, P, F_1)$ and $\mathcal{A}_2 = (Q_2, \Gamma, \rightarrow_2, P, F_2)$

Initialize $R \subseteq Q_1 \times Q_2$ to $\emptyset$
Let $\mathcal{A}_\cap \leftarrow (Q_1 \times Q_2, \Gamma, \rightarrow, \{(p,p) \mid p \in P\}, F_1 \times F_2)$ where $\rightarrow$ initially does not contain any transitions

function AddState(q_1, q_2)
  if $(q_1, q_2) \notin R$ **then**
    $R \leftarrow R \cup (q_1, q_2)$
    if $q_1 \in F_1$ and $q_2 \in F_2$ **then exit and return true**
    for all $q_1' \in Q_1, q_2' \in Q_2, \gamma \in \Gamma$ s.t. $q_1 \xrightarrow{\gamma}_1 q_1'$ and $q_2 \xrightarrow{\gamma}_2 q_2'$ **do**
      add $(q_1, q_2) \xrightarrow{\gamma} (q_1', q_2')$ to $\mathcal{A}_\cap$
      AddState(q_1', q_2')

function AddTransition($q_i \xrightarrow{\gamma}_i q_i'$) ▷ *with* $i \in \{1, 2\}$
  add $q_i \xrightarrow{\gamma}_i q_i'$ to $\mathcal{A}_i$
  for all $q_{3-i}, q_{3-i}' \in Q_{3-i}$ s.t. $(q_1, q_2) \in R$ and $q_{3-i} \xrightarrow{\gamma}_{3-i} q_{3-i}'$ **do**
    add $(q_1, q_2) \xrightarrow{\gamma} (q_1', q_2')$ to $\mathcal{A}_\cap$
    AddState(q_1', q_2')

reachability in pushdown systems using the pre^* and $post^*$ approach described in Section 2. Here only one of the two $\mathcal{P}$-automata is saturated while the other automaton remains unchanged. We now show that this on-the-fly detection of nonemptiness can be applied, with significant performance improvements, also when both approaches are combined.

4.2 Combining Forward and Backward Search

Our experiments show that none of the two approaches, pre^* and $post^*$, is superior to the other one. Our aim is to further improve the reachability analysis of pushdown systems by combining these two methods into $dual*$ algorithm. We first observe the following facts.

Proposition 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and regular sets C and C' of its configurations, the following statements are equivalent: a) $c \Rightarrow^* c'$ for some $c \in C$ and $c' \in C'$, b) $C \cap pre^*(C') \neq \emptyset$, c) $post^*(C) \cap C' \neq \emptyset$, and d) $post^*(C) \cap pre^*(C') \neq \emptyset$.*

Let the $\mathcal{P}$-automata $\mathcal{A}$ and $\mathcal{A}'$ represent the sets of configurations C and C', respectively. The classical approach to the reachability problem, formulated in Proposition 2a, either uses the equivalent formulation in b) and iteratively constructs $\mathcal{A}'_{pre^*}$ while checking whether its language has a nonempty inter-

Algorithm 2 Dual search

Input: $\mathcal{P}$-automata $\mathcal{A}$ and $\mathcal{A}'$

1: **for** p in P **do** AddState(p, p)
2: Initialize pre^* algorithm for $\mathcal{A}'$ and $post^*$ for $\mathcal{A}$ (incl. $workset$s of transitions)
3: **while** $workset_{pre^*} \neq \emptyset$ and $workset_{post^*} \neq \emptyset$ **do**
4: pop t from $workset_{pre^*}$
5: execute one step of pre^* using t
6: **for** t' newly added to $workset_{pre^*}$ **do**
7: AddTransition(t') ▷ *can return* **true**
8: pop t from $workset_{post^*}$
9: execute one step of $post^*$ using t
10: **for** t' newly added to $workset_{post^*}$ **do**
11: AddTransition(t') ▷ *can return* **true**
12: **return false**

section with the set C, or it uses part c) and constructs $\mathcal{A}_{post^*}$ while checking for nonempty intersection with C'.

We suggest a novel combination of these two methods while relying on Proposition 2d. In Algorithm 2, we (sequentially) interleave the executions of the $post^*$ saturation procedure on $\mathcal{A}$ and the pre^* procedure on $\mathcal{A}'$. The intersection of the two automata is computed on-the-fly using Algorithm 1 where each of the saturation procedures calls its respective AddTransition function and Algorithm 2 terminates with **true** as soon as the intersection becomes nonempty. Once one of the saturation algorithms completes its execution, the algorithm returns **false**. Notice that this approach is different from running pre^* and $post^*$ independently in parallel since our algorithm allows the two search directions to 'meet in the middle'. In Section 5 we document a gain of almost an order of magnitude compared to saturating exclusively $\mathcal{A}$ or $\mathcal{A}'$.

4.3 Abstraction Refinement for Pushdown System Reachability

We now explore an abstraction technique [19] in order to reduce the size of the verified PDS. We suggest (in a heuristic way) an initial abstraction by collapsing selected stack symbols and control states and use counter-example guided abstraction refinement [13] in case we obtain spurious traces.

Abstraction of Pushdown Model of MPLS Network. As described in Section 3.2, we consider a network $N = (V, E, s, t, L, \tau)$, NFAs that originate from the given query $\mathcal{N}_a = (S_a, L, \rightarrow_a, s_a, F_a)$, $\mathcal{N}_b = (S_b, E, \rightarrow_b, s_b, F_b)$ and $\mathcal{N}_c = (S_c, L, \rightarrow_c, s_c, F_c)$, and the overapproximating routing table τ^k.

Let $\mathbb{L}$ and $\mathbb{E}$ be the sets of abstract labels resp. edges that are possibly smaller than the sets L and E. A *network abstraction* is a surjective function $\alpha : L \cup E \to \mathbb{L} \cup \mathbb{E}$ such that $\alpha(\ell) \in \mathbb{L}$ for all $\ell \in L$ and $\alpha(e) \in \mathbb{E}$ for all $e \in E$.

Example 1. Let $\mathbb{L} = \{\bullet\}$ and $\mathbb{E} = \{\star\}$ such that $\alpha(\ell) = \bullet$ for $\ell \in L$ and $\alpha(e) = \star$ for $e \in E$. This is the coarsest abstraction that does not distinguish between any labels nor edges. On the other hand, if $\mathbb{L} = L$ and $\mathbb{E} = E$ then the abstraction $\alpha(x) = x$ for $x \in L \cup E$ is the most fine-grained one.

We extend α in a straightforward way to apply to headers and sequences of MPLS operations. We now construct an α-abstracted PDS $\mathcal{P} = (P, \mathbb{L}_\bot, \Delta)$ similar to Section 3.2 such that $P = \mathbb{E} \times \overline{\mathbb{O}ps} \times S_b$ where $\overline{\mathbb{O}ps} = \{\alpha(\omega) \mid \omega \in \overline{Ops}\}$ and Δ is defined as above except that rule of type a) now uses the abstraction:

a) $\langle(\alpha(e), \varepsilon, s), \alpha(\ell)\rangle \hookrightarrow \langle(\alpha(e'), \alpha(\omega), s'), [\alpha(op)](\alpha(\ell))\rangle$ if $(e', op \circ \omega) \in \tau^k(u)$ and $s \xrightarrow{e'}_b{}^* s'$ where either (i) $u = (e, \ell)$, or (ii) $u = e$ and $\ell \in L$, or (iii) $op = \mathtt{push}(\ell')$, $u = e$ and $\ell = \bot$.

We also define α-abstracted initial states $P_i = \{(\alpha(e), \varepsilon, s) \mid e \in E, s \in S_b, s_b \xrightarrow{e}_b{}^* s\}$ and final states $P_f = \{(\alpha(e), \varepsilon, s_f) \mid e \in E,\ s_f \in F_b\}$. Finally, we define an abstraction of an NFA $\mathcal{N} = (S, L, \to, \{s_0\}, F)$ as $\alpha(\mathcal{N}) = (S, \mathbb{L}, \to_\alpha, \{s_0\}, F)$ where $s \xrightarrow{\alpha(\ell)}_\alpha s'$ in $\alpha(\mathcal{N})$ iff $s \xrightarrow{\ell} s'$ in $\mathcal{N}$. Using this, let $\mathcal{N}_i = \alpha(\mathcal{N}_a) \circ \mathcal{N}_\bot$ and $\mathcal{N}_f = \alpha(\mathcal{N}_c) \circ \mathcal{N}_\bot$. Theorem 3 can now be shown to hold also for this α-abstracted PDS.

We now show how to reconstruct a concrete network trace from the α-abstracted pushdown trace. The reconstruction may finish with a success (a concrete network trace is found) or it suggests a refinement of the abstraction function α and the whole verification process is repeated (CEGAR).

Reconstruction of Network Traces. Given a trace $\langle p_0, w_0\rangle \overset{r_1}{\Rightarrow}_\mathcal{P} \ldots \overset{r_m}{\Rightarrow}_\mathcal{P} \langle p_m, w_m\rangle$ in the α-abstracted PDS, we take the subsequence of rules in the trace of type a), and for each such rule r_i define T_i as the set of forwarding rules (u, e', ω) such that r_i was added due to $(e', \omega) \in \tau^k(u)$.

For each set T_i, define $[T_i]$ as a mapping between sets of link-header pairs: $[T_i](C) = \bigcup_{(e,h)\in C}\{(e', h') \mid (u, e', \omega) \in T_i,\ \mathcal{H}(h, \omega) = h', \text{and } u = e \text{ or } u = (e, head(h))\}$. If $C' = [T_i](C)$ then we write $C \underset{T_i}{\Longrightarrow} C'$. The initial set of link-header pairs is $C_0 = \{(e, h) \in E \times L^* \mid p_0 = (\alpha(e), \varepsilon, s),\ s_b \xrightarrow{e}_b{}^* s,\ w_0 = \alpha(h) \circ \bot,\ h \in Lang(\mathcal{N}_a)\}$. The set of reachable link-header pairs is now found by $C_0 \underset{T_1}{\Longrightarrow} C_1 \underset{T_2}{\Longrightarrow} \ldots \underset{T_n}{\Longrightarrow} C_n$. If $C_n \neq \emptyset$ and there exists $(e, h) \in C_n$ such that $h \in Lang(\mathcal{N}_c)$, then we have a concrete network trace, where we finally compute and check the set of failed links against the trace as in Section 3.2. Otherwise the PDS trace is a spurious counter-example that will guide the refinement of the abstraction α.

Refinement from pushdown system rules. If $C_n = \emptyset$ then we compute the refinement based on the rules of the pushdown system: let i be such that $C_i \neq \emptyset$ and $C_{i+1} = \emptyset$, and we must have some $(e, h) \in C_i$ and $(u, e'', \omega) \in T_{i+1}$ such that $u = (e', \ell')$ and $head(h) = \ell$ where $(\alpha(e), \alpha(\ell)) = (\alpha(e'), \alpha(\ell'))$ but $(e, \ell) \neq (e', \ell')$, or that $u = e'$ where $\alpha(e) = \alpha(e')$ but $e \neq e'$. In the refined abstraction α' we ensure that for all such $(e, \ell) \neq (e', \ell')$ we have $(\alpha'(e), \alpha'(\ell)) \neq (\alpha'(e'), \alpha'(\ell'))$, and similarly for such $e \neq e'$ we have $\alpha'(e) \neq \alpha'(e')$. The refined abstraction α' should preferably be as coarse as possible. In the appendix, we present a greedy algorithm (used in our experiments) for computing one such suitable refinement.

Refinement from final headers. If $C_n \neq \emptyset$ but for all $(e, h) \in C_n$ we have $h \notin Lang(\mathcal{N}_c)$ then we compute the refinement based on the transitions in the NFA encoding the final headers: for all pairs $(e, h) \in C_n$ we must have $\alpha(h) \in Lang(\alpha(\mathcal{N}_c))$ but $h \notin Lang(\mathcal{N}_c)$. That is we have in $\alpha(\mathcal{N}_c)$: $s_c \xrightarrow{\alpha(\ell_1)}_\alpha s_1 \xrightarrow{\alpha(\ell_2)}_\alpha \ldots \xrightarrow{\alpha(\ell_n)}_\alpha s_n$ with $h = \ell_1 \ell_2 \ldots \ell_n$, but in $\mathcal{N}_c$: $s_c \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_i} s_i \not\xrightarrow{\ell_{i+1}}$, for some i with $i < n$. Now there must be another label ℓ' such that $\alpha(\ell_{i+1}) = \alpha(\ell')$ and $s_i \xrightarrow{\ell'} s_{i+1}$, but $\ell_{i+1} \neq \ell'$. In the refined abstraction α' we ensure that for all such ℓ' we have $\alpha'(\ell_{i+1}) \neq \alpha'(\ell')$ and we do this for all relevant h.

Heuristics for initial abstraction. We use a heuristic to construct the initial abstraction. We group labels based on their next-hop links, i.e. $\mathbb{L} \subseteq 2^E$ and $\alpha(\ell) = \{e' \mid (e', \omega) \in \tau^k(e, \ell) \text{ for some } e\}$. We group links based on their explicit mention in the path expression of the query, i.e. $\mathbb{E} \subseteq 2^{S_b \times S_b}$ and $\alpha(e) = \{(s, s') \mid s \xrightarrow{e}_b s'\}$.

5 Implementation and Experiments

We implemented the translation of MPLS networks to pushdown automata as well as the three improvements to the reachability analysis in our prototype tool written in C++. In our experimental evaluation, we use real-world network topologies from the Internet Topology Zoo [20]. We implemented a Python tool that for a given network topology distributes the MPLS labels and configures the forwarding tables by following the commonly used Label Distribution Protocol (LDP), the Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE), as well as the industry-standard MPLS VPN services. We generate the forwarding tables using four different parameter settings for the ten largest topologies from [20] (ranging from 100 nodes up to 700 nodes). This results in 40 MPLS data planes, each with 1,520 queries that are randomly instantiated from a set of query templates describing reachability, waypointing, loop-freedom, service-chaining and transparency [14], with the

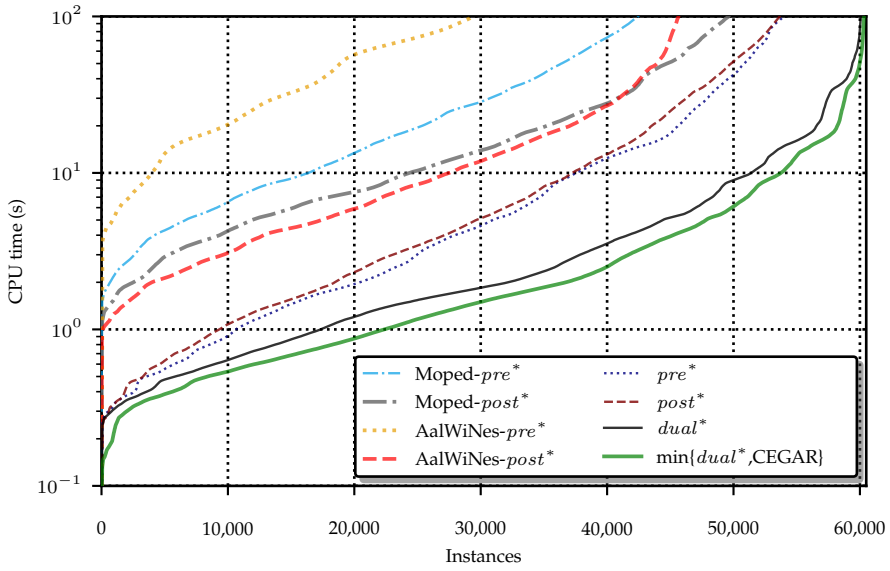


Fig. 2: Comparison of solvers; all 60,800 instances (x-axis) are for each solver independently sorted by the verification time (y-axis, note the logarithmic scale).

maximum number of failures $k \in \{0, 1, 2, 3\}$. We balance the benchmark in order to obtain an even distribution between positive and negative queries. The whole benchmark consists of 60,800 queries that are verified by each of the solvers, in particular our algorithms referred to as $post^*$, pre^* and $dual^*$ (all without CEGAR), compared to the state-of-the-art pushdown reachability algorithms implemented in Moped [4] (Moped-pre^* and Moped-$post^*$) and in AalWiNes [15] (AalWiNes-pre^* and AalWiNes-$post^*$). The experiments were run on a cluster with AMD EPYC 7551 processors at 2.55 GHz (boost disabled) with 32GB memory limit and 100 second timeout. Time spent on parsing files is excluded. The source code, experimental benchmark and all data are available at `https://doi.org/10.5281/zenodo.5005893`.

The results are presented in Figure 2 in terms of performance plots where all instances for the competing approaches are independently sorted by their running times and plotted on the x-axis while the y-axis contains (on logarithmic scale) the respective running times in seconds.

The performance curve for AalWiNes-pre^* and Moped-pre^* are significantly slower than the other methods, including Moped-$post^*$ and AalWines-$post^*$, which are comparable. Our new improved pre^* and $post^*$ methods are comparable performance-wise and already more than two times faster (on the median instance) compared to AalWiNes-$post^*$. This is mainly due to our early termination improvement and a more efficient encoding of the network.

Topology	Query	CEGAR	$dual^*$	Speedup
Colt	$\langle \cdot^* \rangle\ [\cdot \# Toulouse]\ [\hat{}\ \cdot \# Milan, \cdot \# Poit]^*\ [Bari\# \cdot]\ \langle \cdot^* \rangle\ 0$	0.94	90.54	96.42
Pern	$\langle \cdot^* \rangle\ [\cdot \# N56]\ [\hat{}\ \cdot \# N38, \cdot \# Isla, \cdot \# N54]^*\ [N99\# \cdot]\ \langle \cdot^* \rangle\ 0$	0.35	34.30	97.10
Colt	$\langle \cdot \rangle\ [\cdot \# Paris]\ \cdot^*\ [Livorno\# \cdot]\ \langle \cdot^+ \cdot \rangle\ 0$	1.00	>100.00	>100.00
Colt	$\langle \cdot ? \rangle\ [\cdot \# Strasbourg]\ \cdot^*\ [\cdot \# Piacenza]\ \cdot^*\ [Novara\# \cdot]\ \langle \cdot ? \rangle\ 0$	1.00	>100.00	>100.00
Colt	$\langle \cdot ? \rangle\ [\cdot \# Karlsruhe]\ \cdot^*\ [\cdot \# Ostend]\ \cdot^*\ [Brindisi\# \cdot]\ \langle \cdot ? \rangle\ 0$	0.98	>100.00	>102.04

Fig. 3: The queries that perform relatively best for CEGAR (time in seconds)

The introduction of our $dual^*$ approach significantly improves the performance of both pre^* and $post^*$, and on the median instance the $dual^*$ solver is more than 6 times faster than the previous state-of-the-art AalWiNes-$post^*$ approach, while the curves further open with the increasing complexity of the reachability problems. On the instance number 49,629 (the largest instance that Moped-$post^*$ solved) $dual^*$ is already 11 times faster than Moped-$post^*$. With the harder instances $dual^*$ performs increasingly better than both pre^* and $post^*$.

The performance of the CEGAR approach is incomparable with $dual^*$ as on 27% of all instances CEGAR is faster (sometimes even by two orders of magnitude) but on the remaining instances it can be significantly slower. We noticed that the CEGAR approach is considerably better performing on negative queries (without any trace) where it is faster on 47% cases. The best cases for CEGAR with two orders of magnitude speedup are listed in Figure 3 and we remark that CEGAR solved 249 queries where $dual^*$ timed out. The number of CEGAR iterations where the method is faster than $dual^*$ ranges between 1 to 61 but typically less than 10 iterations are required to get a conclusive answer. As $dual^*$ and CEGAR are incomparable, we use the pragmatic approach where we can run both of them in parallel and terminate as soon as one of the methods provides an answer. This is depicted by the curve $\min\{dual^*, \text{CEGAR}\}$ that further improves the performance by additional 20–30%. In particular this combined method is 7.5 times faster than AalWiNes-$post^*$ on the median case and 17 times faster than Moped-$post^*$ on the instance number 49,629.

Finally, as both the network encoding in AalWiNes [15] as well as in our paper overapproximate the set of network traces, they can provide inconclusive answers. On our benchmark, AalWiNes-$post^*$ returned 2,024 inconclusive answers, whereas our encoding approach reported only 7 inconclusive answers for $dual^*$ and 6 inconclusive answers for $dual^*$ combined with CEGAR.

6 Conclusion

While more automated approaches to verify and operate communication networks can significantly improve their dependability, this requires efficient algorithms which can deal with the large scale and complexity of today's net-

works. We presented an efficient translation from MPLS routing tables into pushdown systems. We also revisited the problem of fast reachability analysis of pushdown systems and presented three techniques improving the performance over the state-of-the-art solution by an order of magnitude. In the future work we plan to study fast algorithms for verifying quantitative reachability properties (related to latency or network congestion) via weighted pushdown automata.

Acknowledgements. We thank to Bernhard Schrenk for updating the AalWiNes online demo at `https://demo.aalwines.cs.aau.dk` with the improved verification engine described in this paper.

Research supported by the Vienna Science and Technology Fund (WWTF), ICT19-045 (WHATIF), and the DFF project QASNET.

References

[1] J. Esparza and J. Knoop, "An automata-theoretic approach to interprocedural data-flow analysis," in *FOSSACS'99*, ser. LNCS, vol. 1578. Springer, 1999, pp. 14–30.

[2] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *CAV'05*, ser. LNCS, vol. 3576. Springer, 2005, pp. 449–461.

[3] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 324–336.

[4] S. Schwoon, "Moped," in *http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/*, 2002.

[5] D. Suwimonteerabuth, S. Schwoon, and J. Esparza, "jMoped: A java bytecode checker based on Moped," in *TACAS'05*, ser. LNCS, vol. 3440. Springer, 2005, pp. 541–545.

[6] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

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

[8] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *Proc. ACM POPL*, 2014, pp. 113–126.

[9] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012, pp. 113–126.

[10] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *CAV'17*, ser. LNCS, vol. 10427. Springer, 2017, pp. 261–281.

[11] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating MPLS networks," in *Proc. IEEE INFOCOM*, 2018, pp. 1799–1807.

[12] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, 2002.

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 1855. Springer, 2000, pp. 154–169.

[14] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[15] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[16] J. Esparza, S. Kiefer, and S. Schwoon, "Abstraction refinement with Craig interpolation and symbolic pushdown systems," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 5, no. 1-4, pp. 27–56, 2009.

[17] J. R. Büchi, "Regular canonical systems," *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[18] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," in *INFINITY'97*, ser. ENTCS, vol. 9. Elsevier, 1997, pp. 27–37.

[19] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.

[20] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[21] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," RFC 4090, May 2005.

A Appendix

Proof Sketch for Proposition 1

We first prove the following lemma:

Lemma 1. *After initialization, the following invariant holds for any sequence of calls to* AddTransition *in Algorithm 1: for all $p \in P$ we have $(p,p) \xrightarrow{w}{}^*(q_1,q_2)$ iff $p \xrightarrow{w}{}_1^* q_1$ and $p \xrightarrow{w}{}_2^* q_2$, and we have $R = \{(q_1,q_2) \mid \exists p \in P.\ (p,p) \to^* (q_1,q_2)\}$.*

Proof (Sketch). Base: Initially $\to$ and R are empty. If $p \xrightarrow{w}{}_1^* q_1$ and $p \xrightarrow{w}{}_2^* q_2$, then the call AddState(p, p) will (using the recursive depth-first-search) add transitions to $\mathcal{A}_\cap$ such that $(p,p) \xrightarrow{w}{}^*(q_1,q_2)$, and add the state (q_1,q_2) to R (unless the initialization terminates early). No other transitions or states are added.

Invariant preservation: Let R, $\to$, $\to_1$ and $\to_2$ be the values before, and R', $\to'$, $\to_1'$ and $\to_2'$ be the values after a call to AddTransition($q_i \xrightarrow{\gamma}{}_i q_i'$). Consider each matching transition $q_{3-i} \xrightarrow{\gamma}{}_{3-i} q_{3-i}'$ in the other $\mathcal{P}$-automaton. If $(q_1,q_2) \in R$ then for some $p \in P, w \in \Gamma^*$ $(p,p) \xrightarrow{w}{}^*(q_1,q_2)$ and hence $p \xrightarrow{w}{}_1^* q_1$ and $p \xrightarrow{w}{}_2^* q_2$. Due to line 11 and 12 we have $p \xrightarrow{w\gamma}{}_1'^* q_1'$ and $p \xrightarrow{w\gamma}{}_2'^* q_2'$, and on line 13 we get $(p,p) \xrightarrow{w\gamma}{}'^*(q_1',q_2')$. Furthermore, all transitions and states reachable from (q_1',q_2') are added during the call to AddState(q_1', q_2') using depth-first-search of matching transitions in $\to_1'$ and $\to_2'$. If $(q_1,q_2) \notin R$ then for all $p \in P$ $(p,p) \not\to^*(q_1,q_2)$ and hence for all $w \in \Gamma^*$ either $p \xrightarrow{w}{\not\to}_1^* q_1$ or $p \xrightarrow{w}{\not\to}_2^* q_2$. Therefore either $p \xrightarrow{w\gamma}{\not\to}_1'^* q_1'$ or $p \xrightarrow{w\gamma}{\not\to}_2'^* q_2'$, and hence no transitions need to be added to $\mathcal{A}_\cap$ for this match. □

From Lemma 1 and the fact that the final states of $\mathcal{A}_\cap$ are $F_1 \times F_2$ we have that $Lang(\mathcal{A}_\cap) = Lang(\mathcal{A}_1') \cap Lang(\mathcal{A}_2')$.

From $R = \{(q_1,q_2) \mid \exists p \in P.\ (p,p) \to^* (q_1,q_2)\}$ as per Lemma 1, and the fact that line 6 of Algorithm 1, immediately after a state is added to R, checks whether it is a final state, we see that as soon as $Lang(\mathcal{A}_\cap) \neq \emptyset$, the algorithm returns true. This completes the proof of Proposition 1. □

Search Efficiency of CEGAR

The sets of possible link-header pairs, in particular C_0, can be very large, so we use two techniques to succinctly represent the search states. First, we use depth first search of the configuration sets $C_0 \underset{T_1}{\Longrightarrow} \ldots \underset{T_n}{\Longrightarrow} C_n$ to avoid storing most states in memory and to enable early termination if a valid reconstruction is found. We keep track of the current deepest C_i, for potentially computing refinement based on pushdown system rules. Second, we succinctly represent all headers in C_0 by a stack of wildcards with the correct size. When we

follow a forwarding rule $t \in T_i$ during the depth first search, we specialize the wildcard to the required precondition label ℓ_i for that rule. We know the accepting path in $\alpha(\mathcal{N}_a)$: $s_a \ldots \xrightarrow{\alpha(\ell_i)}_\alpha s_i \ldots$, so we check that in $\mathcal{N}_a$: $s_{i-1} \xrightarrow{\ell_i} s_i$ which eventually ensures that $h_0 \in \mathit{Lang}(\mathcal{N}_a)$. If there are still wildcards left, when we reach the final header h_n, we follow the remaining transitions from both $\mathcal{N}_a$ and $\mathcal{N}_c$ in lockstep to find concrete labels such that both $h_0 \in \mathit{Lang}(\mathcal{N}_a)$ and $h_n \in \mathit{Lang}(\mathcal{N}_c)$ are satisfied. If this is not possible, we have a spurious counter-example and find a refinement based on this. Finally, the search keeps track of the used and failed links, and avoids searching down branches, where it is already clear that the final check $\{e_0, \ldots, e_n\} \cap F = \emptyset$ and $|F| \leq k$ will fail.

Computing a Small Pair Refinement

For the CEGAR approach, we need a way of computing a refinement based on a spurious counter-example. The refinement should remove this spurious counter-example, while not making the next α-abstraction too fine-grained, since that would increase the size of PDS. The case where we have pairs $(\alpha(e), \alpha(\ell)) = (\alpha(e'), \alpha(\ell'))$ but $(e, \ell) \neq (e', \ell')$ turns out to be non-trivial. Here we abstract away from the use in CEGAR and define the problem of computing a small pair refinement as follows.

Given sets A and B, and sets of pairs $X \subseteq A \times B$ and $Y \subseteq A \times B$, find partitionings $A_1 \uplus \cdots \uplus A_n = A$ and $B_1 \uplus \cdots \uplus B_m = B$ with a) minimal n and m, such that b) for all i, j, $1 \leq i \leq n$, $1 \leq j \leq m$ we have $X \cap (A_i \times B_j) = \emptyset \vee Y \cap (A_i \times B_j) = \emptyset$.

We wish to avoid any i, j with $X \cap (A_i \times B_j) \neq \emptyset$ and $Y \cap (A_i \times B_j) \neq \emptyset$, since that would (locally) allow the spurious counter-example to reappear in the next iteration of CEGAR. Secondly, we wish to minimize n and m since this will keep the refinement as small as possible.

Since performance is important for this application, we present a greedy algorithm that satisfies b), while it relaxes a) to only small, rather than minimal, values for n and m.

We introduce the following notation: Functions $X_A(b) = \{a \mid (a, b) \in X\}$, $X_B(a) = \{b \mid (a, b) \in X\}$, $Y_A(b) = \{a \mid (a, b) \in Y\}$, $Y_B(a) = \{b \mid (a, b) \in Y\}$. Sets $X_A = \bigcup_{b \in B} X_A(b)$, $X_B = \bigcup_{a \in A} X_B(a)$, $Y_A = \bigcup_{b \in B} Y_A(b)$, $Y_B = \bigcup_{a \in A} Y_B(a)$.

Algorithm 3 assigns elements of A and B into buckets in a greedy manner, choosing first buckets for elements of A and then B in a way that ensures condition b) is fulfilled. When needed a new bucket is created.

Details on Tool for Generation of MPLS Data Plane

While the topologies of many real communication networks have been made available online, e.g., [20], this data does not include the router tables required to model MPLS data planes. For this paper, we hence develop a tool which

Algorithm 3 Greedy algorithm for computing pair refinement

Initialize bucket A_1 with $A \setminus (X_A \cup Y_A)$
for each a in $X_A \cup Y_A$ **do**
    for each existing bucket A_i **do**
        Let $Z_X \leftarrow \bigcup_{a' \in A_i} X_B(a')$, and $Z_Y \leftarrow \bigcup_{a' \in A_i} Y_B(a')$
        if $(X_B(a) \cap Z_Y) \cup (Y_B(a) \cap Z_X) = \emptyset$ **then**
            Put a in bucket A_i
    if a was not assigned to a bucket **then**
        Initialize new bucket with a
Initialize bucket B_1 with $B \setminus (X_B \cup Y_B)$
for each b in $X_B \cup Y_B$ **do**
    for each existing bucket B_i **do**
        Let $Z_X \leftarrow \bigcup_{b' \in B_i} X_A(b')$, and $Z_Y \leftarrow \bigcup_{b' \in B_i} Y_A(b')$
        if $\forall (a, a') \in (X_A(b) \times Z_Y) \cup (Y_A(b) \times Z_X)$, a and a' are not in the same bucket **then**
            Put b in bucket B_i
    if b was not assigned to a bucket **then**
        Initialize new bucket with b
return all buckets $A_1 \dots A_n$ and $B_1 \dots B_m$

allows to generate, for a given network topology, realistic synthetic data planes. Concretely, given a network topology (with weighted links as expected from an IGP topology), our tool directly computes the MPLS data plane that will be obtained after running typically deployed MPLS protocols Label Distribution Protocol (LDP)[1] and Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE)[2] until convergence. The tool also allows instantiating industry-standard MPLS VPN services. The protocol related parameters can be adjusted for each experiment. By generating the forwarding tables according to usual protocols and practice, the result is a data plane for experimentation that shares similarities on its construction to the ones found on real MPLS networks.

Motivation for Redundant Paths and Labels on MPLS Networks

MPLS networks often feature significant redundancy in paths and labels, which can be exploited for optimization. There are a few reasons that explain why many paths on a MPLS network may have significant overlap or path redundancy. On networks that use LDP, this protocol is in charge of distributing and setting up paths across the network to reach IP prefixes present in routing

[1] See `https://www.rfc-editor.org/rfc/rfc5036.txt`.
[2] See `https://www.rfc-editor.org/rfc/rfc3209.txt`.

tables of different routers. A typical strategy is to allocate a single MPLS label to all prefixes that can be reached through the same BGP next-hop[3]. Thus if the LDP process of a router at the edge of the MPLS domain creates a path tree for each of its BGP next-hops (a common situation when connecting with other networks), then the result is a network containing several labels along exactly the same paths, hence the redundancy. Also, LDP can be configured to introduce further deaggregation resulting in further redundancy[4] Another possibility is due to having a few RSVP tunnel tailends concentrating many tunnels from different headends. In this case, a concentration of paths tend to appear when getting closer to the destination. This is not rare in practice for MPLS domain edge nodes connecting to IXPs or datacenters. The result is having many labels pointing to the same interface, or even more, to the same paths or common segments of paths, introducing a redundancy in the forwarding. Yet another situation on which path redundancy may arise is due to usage of One-to-One MPLS Fast Re Route protections [21]. In this case, given a RSVP path across the network, each router on said path computes an alternative path to forward packets in the event of link failure upstream, eventually merging with the original path. This protection might effectively multiply the number of forwarding paths in the network by a factor proportional to the average path length, increasing path redundancy if the ratio of tunnels to paths in the network is already high.

[3]Juniper LDP overview `https://www.juniper.net/documentation/us/en/software/junos/mpls/topics/topic-map/ldp-overview.html`

[4]MPLS LDP FEC deaggregation `https://www.juniper.net/documentation/en_US/junose15.1/topics/task/configuration/mpls-ldp-fec-deaggregation.html`

Paper D

AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks

Peter Gjøl Jensen, Dan Kristiansen,
Stefan Schmid, Morten Konggaard Schou,
Bernhard Clemens Schrenk, and Jiří Srba

The paper has been published in:
Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), pp. 474-481, Association for Computing Machinery, 2020. https://doi.org/10.1145/3386367.3431308

The layout has been revised.

Abstract

We present an automated what-if analysis tool AalWiNes for MPLS networks which allows us to verify both logical properties (e.g., related to the policy compliance) as well as quantitative properties (e.g., concerning the latency) under multiple link failures. Our tool relies on weighted pushdown automata, a quantitative extension of classic automata theory, and takes into account the actual data plane configuration, rendering it especially useful for debugging. In particular, our tool collects the different router forwarding tables and then builds a pushdown system, on which quantitative reachability is performed based on an expressive query language. Our experiments show that our tool outperforms state-of-the-art approaches (which until now have been restricted to logical properties) by several orders of magnitude; furthermore, our quantitative extension only entails a moderate overhead in terms of runtime. The tool comes with a platform-independent user interface and is publicly available as open-source, together with all other experimental artefacts.

1 Introduction

While communication networks are a critical infrastructure of our digital society, their correct configuration and operation is complex, requiring operators to become "*masters of complexity*" [1]. As many recent network outages were caused by human errors, e.g., [2–4], we currently witness major research efforts toward more automated and formal approaches to operate and verify networks [5–13].

A particularly critical but challenging task for human operators is to *reason about failures* (in this paper referred to as *what-if analysis*). In order to meet their stringent dependability requirements, most modern communication networks come with fast recovery mechanisms which revert traffic to alternative paths [14–17]. While this is attractive, already a single link failure can lead to unintended network behaviors which are easily overlooked and may violate the network policy [4]. Especially multiple link failures, which are more likely to occur in large networks and can be caused e.g. due to shared risk link groups [18–20], may threaten network dependability.

It is often insufficient to only ensure the logical correctness (e.g., policy compliance) of the network behavior under failures. A dependable network also needs to satisfy quantitative properties. For example, traffic should be rerouted along *short* paths, e.g., regarding link latency (offering a low latency) or number of hops (reducing load), even under a certain number of link failures.

We are particularly interested in networks based on Multiprotocol Label Switching (MPLS) [21]. MPLS networks are widely deployed in the Internet today, especially in IP networks and for traffic engineering purposes. The

study of MPLS networks is also interesting from a theoretical perspective, as the header size in these networks is not fixed; rather, additional labels may be pushed on the header while rerouting packets around failed links, creating "tunnels". This makes the employment of formal methods particularly challenging as we must deal with a possibly unbounded set of packet headers.

Our Contributions. We present a what-if analysis tool for MPLS networks, AalWiNes[1], which supports a fully automated and fast verification of the network behavior under failures. In particular, AalWiNes relies on an expressive query language and allows us to test both logical properties (such as the policy compliance) as well as quantitative properties (such as the latency, number of hops, required label stack size resp. tunneling depth, or number of failed links), and in polynomial-time using an over- and under-approximation technique for an arbitrary number of link failures (and with a low number of inconclusive answers). AalWiNes operates directly on the dataplane forwarding tables, allowing to debug issues not visible in the control plane.

At the heart of AalWiNes lies a *weighted* pushdown automaton, a quantitative generalization of classical automata: based on the router forwarding tables and a query (the input to the tool), we build a weighted pushdown system and then perform a quantitative reachability analysis. To improve performance, AalWiNes uses novel algorithms tailored to this use case.

We offer an optimized C++ implementation of AalWiNes and report on its platform-independent user interface. Considering a case study in cooperation with NORDUnet, a major network operator, we show that our tool outperforms state-of-the-art approaches (only applicable to verification of logical properties) by several orders of magnitude in terms of runtime. We also demonstrate that our quantitative extension only entails a reasonable overhead. As a contribution to the research community and in order to ensure reproducibility, we released our tool as open source and we also shared all our experimental artefacts [22].

Related work and novelty. The problem of how to render networks more automated and formally verifiable has recently received much interest, both for specific networks and protocols, such as BGP [23], OpenFlow [5, 12], or MPLS [7] networks, as well as for networks which are protocol agnostic [6]. The different systems rely on different approaches, including e.g., algebraic approaches [5], static verification based on geometric approaches [6], or automata-theoretic approaches [24]. We specifically consider MPLS networks and use an automata-theoretic approach. Whereas some recent work focus on verification of the control plane configurations [11, 25–29], we directly verify the router forwarding tables, which allows us to catch errors in the data plane [30].

We focus on *polynomial-time* verification via a suitable over- and under-

[1] AALborg WIen NEtwork verification Suit

approximation, even under failures: many existing approaches in the literature do not consider failure scenarios explicitly (and still exhibit a super-polynomial runtime, e.g., due to SAT solving [10]), and/or have to solve NP-hard problems when modelling failures scenarios, e.g. SMT solving [11]. Furthermore, most existing approaches target different network types [11, 25, 26, 28, 29] and do not support arbitrary header sizes, which however arise in the context of MPLS networks: by representing MPLS networks symbolically as pushdown automata, we hence achieve an exponential speedup compared to the direct encoding of all possible sequences of header symbols.

To the best of our knowledge, our tool is the first to consider what-if analysis of quantitative aspects as well, and we are not aware of any applications of weighted automata theoretical results in this context. The few weighted solutions that exist do not consider failure scenarios and have an exponential runtime [31]. The paper closest to ours is P-Rex [24], a polynomial-time approach to verify logical properties of MPLS networks, also accounting for possible failures by using approximative analysis. As we demonstrate in our evaluation, our tool not only adds the novel quantitative dimension, but also outperforms P-Rex by several orders of magnitude. We further contribute an interactive user interface and make a leap forward regarding applicability for network operators.

Finally, we note that what-if analyses tools have been developed also various other networking contexts, such as in CDNs, to predict the effects of possible configuration and deployment changes [32].

2 MPLS Network Model

This section introduces our MPLS model and query language.

2.1 Network definition

An MPLS network consists of a topology and forwarding rules.

Definition 1. A *network topology* is a directed multigraph (V, E, s, t) where V is a set of *routers*, E is a set of *links* between routers, $s : E \to V$ assigns the *source router* to each link, and $t : E \to V$ assigns the *target router*.

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. A link is *active* if it belongs to $E \setminus F$. We assume asymmetric link failures that can be caused e.g., by congestion in one direction, resulting in packet drops that can also appear as a link failure.

Let L be a nonempty set of MPLS labels used in packet headers. We define the set of MPLS operations on packet headers as $Op = \{\texttt{swap}(\ell) \mid \ell \in$

$L\} \cup \{\texttt{push}(\ell) \mid \ell \in L\} \cup \{\texttt{pop}\}$. Given an alphabet A, let A^* denote the set of all finite words over the elements of A, including the empty word ϵ.

Definition 2. An *MPLS network* is a tuple $N = (V, E, s, t, L, \tau)$ where (V, E, s, t) is a network topology, $L = L_M \uplus L_M^{\perp} \uplus L_{IP}$ is a finite set of labels partitioned into (1) the MPLS label set L_M, (2) the set of MPLS labels with the bottom of the stack bit (S) set to true $L_M^{\perp}$, (3) a set of IP addresses L_{IP}, and $\tau : E \times L \to (2^{E \times Op^*})^*$ is the routing table.

The routing table, for every link $e \in E$ and a top (left-most) packet label ℓ, returns a sequence of traffic engineering groups $\tau(e, \ell) = O_1 O_2 \ldots O_n$ where each traffic engineering group is a set of the form $\{(e_1, \omega_1), \ldots, (e_m, \omega_m)\}$ where e_j is the outgoing link such that $t(e) = s(e_j)$ and $\omega_j \in Op^*$ is a sequence of operations to be performed on the packet header. In a given traffic engineering group, the router can nondeterministically (e.g. pseudorandomly) select any active link and forward the packet via that link while applying the corresponding sequence of MPLS operations. This allows us to abstract away from various routing policies that facilitate e.g. splitting of a flow along multiple shortest paths. The group O_i has a higher priority than O_{i+1} and during the forwarding, and the router always selects the traffic engineering group with the highest priority and at least one active link.

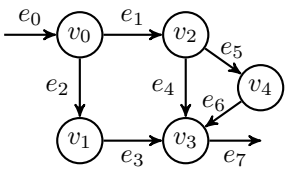
2.2 Valid MPLS headers

The MPLS labels can be nested only in a specific way. For a given network $N = (V, E, s, t, L, \tau)$, we define the set of valid headers $H \subseteq L^*$ by $H = L_{IP} \cup \{\alpha \ell_1 \ell_0 \mid \alpha \in L_M^*,\ \ell_1 \in L_M^{\perp},\ \ell_0 \in L_{IP}\}$. Hence the on top of the IP label there can be one label with the bottom of stack bit S set to true and an arbitrary number of other MPLS labels. The MPLS operations can manipulate the label-stack by modifying only the topmost label so that the result of operations performed on a valid header is itself a valid header.

Definition 3. The semantics of MPLS operations is a partial *header rewrite function* $\mathcal{H} : H \times Op^* \hookrightarrow H$ where $\omega, \omega' \in Op^*$, $\ell \in L$, $h \in H$ and ϵ is the empty sequence of operations:

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

Let $L_M = \{30, 31\}$, $L_M^{\perp} = \{s20, s21\}$ and $L_{IP} = \{ip_1\}$. We use here and in what follows the convention that all labels that are on the bottom of the MPLS



(a) Network topology

Router	e_{in}	Label	Priority	e_{out}	Operation
v_0	e_0	ip_1	1	e_1	$\texttt{push}(s20)$
	e_0	ip_1	1	e_2	$\texttt{push}(s10)$
	e_0	$s40$	1	e_1	$\texttt{swap}(s41)$
v_1	e_2	$s10$	1	e_3	$\texttt{swap}(s11)$
v_2	e_1	$s20$	1	e_4	$\texttt{swap}(s21)$
	e_1	$s41$	1	e_5	$\texttt{swap}(s42)$
	e_1	$s20$	2	e_5	$\texttt{swap}(s21) \circ \texttt{push}(30)$
v_3	e_3	$s11$	1	e_7	$\texttt{pop}$
	e_4	$s21$	1	e_7	$\texttt{pop}$
	e_6	$s43$	1	e_7	$\texttt{swap}(s44)$
	e_6	$s21$	1	e_7	$\texttt{pop}$
v_4	e_5	30	1	e_6	$\texttt{pop}$
	e_5	$s42$	1	e_6	$\texttt{swap}(s43)$

(b) Routing table

$\sigma_0 = (e_0, ip_1)\ (e_1, s20 \circ ip_1)\ (e_4, s21 \circ ip_1)\ (e_7, ip_1)$
$\sigma_1 = (e_0, ip_1)\ (e_2, s10 \circ ip_1)\ (e_3, s11 \circ ip_1)\ (e_7, ip_1)$
$\sigma_2 = (e_0, ip_1)\ (e_1, s20 \circ ip_1)\ (e_5, 30 \circ s21 \circ ip_1)\ (e_6, s21 \circ ip_1)\ (e_7, ip_1)$
$\sigma_3 = (e_0, s40 \circ ip_1)\ (e_1, s41 \circ ip_1)\ (e_5, s42 \circ ip_1)\ (e_6, s43 \circ ip_1)\ (e_7, s44 \circ ip_1)$

(c) Examples of traces

Query	Witness traces
$\varphi_0 = \langle \texttt{ip} \rangle\ [.\#v_0]\ .^*\ [v_3\#.]\ \langle \texttt{ip} \rangle\ 0$	σ_0, σ_1
$\varphi_1 = \langle \texttt{ip} \rangle\ [.\#v_0]\ [\hat{}\ v_2\#v_3]^*\ [v_3\#.]\ \langle \texttt{ip} \rangle\ 2$	σ_1, σ_2
$\varphi_2 = \langle s40\ \texttt{ip} \rangle\ [.\#v_0]\ .^*[v_3\#.]\ \langle \texttt{smpls ip} \rangle\ 0$	σ_3
$\varphi_3 = \langle s40\ \texttt{ip} \rangle\ [.\#v_0]\ .^*[v_3\#.]\ \langle \texttt{mpls}^+\ \texttt{smpls ip} \rangle\ 1$	no trace exists
$\varphi_4 = \langle \texttt{smpls?}\ \texttt{ip} \rangle\ [.\#v_0]\ \ldots .^*[v_3\#.]\ \langle \texttt{smpls?}\ \texttt{ip} \rangle\ 1$	σ_2, σ_3

(d) Network queries

Fig. 1: A small network example

label stack (have the bottom of stack bit S set to true) are prefixed with small s. Then $\mathcal{H}(30 \circ s20 \circ ip_1,\ \texttt{pop} \circ \texttt{swap}(s21) \circ \texttt{push}(31)) = 31 \circ s21 \circ ip_1$.

2.3 Example network

An example of a simple network topology is given in Figure 1a together with the forwarding table described in Figure 1b. The example defines two label switching paths for IP-packet routing from v_0 to v_3, either via the links e_1 and e_4, or the links e_2 and e_3. The respective path can be selected nondeterministically. Moreover, packets arriving via the link e_0 with the service label $s40$ (agreement with the neighboring network operator) are routed via the links e_1, e_5, e_6 and leave the network on the link e_7.

Every forwarding rule for the router v is represented by a line in the table and depending on the incoming link e_{in} (where $t(e_{in}) = v$) and the top of

the stack label, it determines the outgoing link e_{out} (where $s(e_{out}) = v$) and a sequence of stack operations that replace the top label. Each such rule has a priority that is depicted by the priority column in the middle of the table. In our example, it is only the router v_2 that has more than one priority group associated to its forwarding table in order to protect the link e_4. If a packet arrives via the link e_1 with label $s20$ on top of the stack then it is primarily forwarded via the link e_4 while the label is swapped with $s21$. Only if the link e_4 fails, a backup rule with priority 2 is used so that it forwards the traffic via the link e_5, first swapping the top label with $s21$ and then pushing a new label 30 on top of the label stack. The router v_4 then pops the label and the packet arrives to v_3 with the same label as if the link e_4 did not fail.

2.4 Network traces

We now define valid traces in an MPLS network $N = (V, E, s, t, L, \tau)$ under the assumption that the links in the set $F \subseteq E$ failed. For a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \ldots, (e_m, \omega_m)\}$ we let $E(O) = \{e_1, e_2, \ldots, e_m\}$ denote the set of all links in the group. The group O is *active* if it contains at least one active link, i.e. $E(O) \setminus F \neq \emptyset$. Further, we define $\mathcal{A}(O_1 O_2 \ldots O_n) = \{(e, \omega) \in O_j \mid e \text{ is an active link}\}$ where j is the lowest index such that O_j is an active traffic engineering group, and we let $\mathcal{A}(O_1 O_2 \ldots O_n) = \emptyset$ if no such j exists. The set $\mathcal{A}(\tau(e, \ell))$ so contains all the currently available output links and the corresponding label-stack operations to be performed on a packet arriving on the link e with the top-most label ℓ. A trace in a network is a routing of a packet in the network and consists of a sequence of active links together with the corresponding label-stack headers.

Definition 4. A *trace* in a network $N = (V, E, s, t, L, \tau)$ with a set of failed links $F \subseteq E$ is any finite sequence

$$(e_1, h_1)(e_2, h_2) \ldots (e_n, h_n) \in \big((E \setminus F) \times H\big)^*$$

of link-header pairs where $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, head(h_i)))$ for all i, $1 \leq i < n$, where $head(h_i)$ is the top (left-most) label of h_i.

Examples of network traces for our running example are provided in Figure 1c. The traces σ_0 and σ_1 describe two possible traces for routing a packet arriving to v_0 with the destination IP ip_1, under the assumption that $F = \emptyset$. The trace σ_2 shows a failover protection of the link between v_2 and v_3 in case that $F = \{e_4\}$. Finally, the trace σ_4 encodes a label switching path for packets arriving to v_0 with the service label $s40$ and it is a valid trace for example for the set of failed links $F = \{e_2, e_3\}$.

2.5 Query language

We present a powerful query language that allows us to specify regular trace properties, both regarding the initial and final label-stacks as well as the link sequence in the trace.

Definition 5. A reachability *query* for an MPLS network $N = (V, E, s, t, L, \tau)$ is of the form $\langle a \rangle\ b\ \langle c \rangle\ k$ where a and c are regular expressions over the set of labels L, b is a regular expression over the set of links E, and $k \geq 0$ specifies the maximum number of failed links to be considered.

We assume here a standard syntax for regular expressions and by $Lang(a)$, $Lang(b)$ and $Lang(c)$ we understand the regular language defined by the expressions a, b and c, respectively. For specifying labels in the regular expressions a and c we use the abbreviations:

- $\texttt{ip} = [ip_0, \ldots, ip_n]$ where $L_{IP} = \{ip_0, \ldots, ip_n\}$,
- $\texttt{mpls} = [\ell_0, \ldots, \ell_n]$ where $L_M = \{\ell_0, \ldots, \ell_n\}$, and
- $\texttt{smpls} = [\ell_0^\perp, \ldots, \ell_n^\perp]$ where $L_M^\perp = \{\ell_0^\perp, \ldots, \ell_n^\perp\}$.

We further use the following notation for specifying links in the network. If v and u are routers, then $[v\#u]$ matches any link e from v to u such that $s(e) = v$ and $t(e) = u$. If in_1 is an interface on router v that uniquely identifies the outgoing link e, and in_2 identifies the incoming interface on router u for the link e, then $[v.in_1\#u.in_2]$ matches exactly the link e. The dot-syntax is used to denote any link in the network and it is extended to match also any router so that $[v\#\cdot] = \bigcup_{u \in V} [v\#u]$ and $[\cdot\#u] = \bigcup_{v \in V} [v\#u]$.

Problem 1 (Query Satisfiability Problem). Given an MPLS network N and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, decide if there exists a trace $\sigma = (e_1, h_1) \ldots (e_n, h_n)$ in the network N for some set of failed links F such that $|F| \leq k$ where $h_1 \in Lang(a)$, $e_1 \ldots e_n \in Lang(b)$, and $h_n \in Lang(c)$. If this is the case, the query φ is *satisfied* and we call σ a *witness trace*.

Examples of queries are provided in Figure 1d. The query φ_0 asks about the existence of a trace that starts and ends with the label-stack containing only the IP header, such that the first link is incoming to the router v_0, followed by zero or more hops via unspecified links, and ending with link that leaves the router v_3, all under the assumption of no link failures. The traces σ_0 and σ_1 satisfy the query, however, even though the trace σ_2 has the required form as well, it does not satisfy φ_0 as it requires that the link e_4 fails. The next query φ_1 expresses a similar property as φ_0 with the exception that we allow for up to 2 link failures and the inner path may not contain any link between v_2 and v_3 (the symbol $\hat{}$ stands for complement of regular expressions). The traces

σ_1 and σ_2 satisfy this query. The query φ_2 asks about a possible routing path between v_0 and v_3 where the header of the initial packet contains the label $s40$ on top of an IP header and leaves the network with an arbitrary MPLS label (where the bottom of the stack bit is set to true) on top of the IP header. Indeed, the trace σ_3 has this property and it is a valid trace even in case of no link failures. The next query φ_3 checks the transparency of the routing from v_0 to v_3 by asking whether a packet with the service label $s40$ can leave our network with at least one additional MPLS label on top of the service label. Should this be the case then our network leaks internal MPLS labels to the neighboring networks, which is not desirable. Even in case of one link failure, the query is not satisfied. Finally, the query φ_4 asks whether in case of one link failure there is an IP routing, with an optional MPLS label on the top of the IP label, with three or more hops between the incoming and outgoing links, and this is indeed the case as documented by the witness traces σ_2 and σ_3. In case of no link failures, the query is satisfied only by the trace σ_3.

3 Quantitative Extension

After describing our network model and the query language used in our tool, we now present a novel extension of the framework which allows us to account for quantitative aspects, like latency, number of hops, tunnels (label stack size), number of failures, and linear combinations of these measures.

For a given network query, there can be several network traces that satisfy the query and for some queries there exist even infinitely many witness traces. From the user perspective, it is hence essential that when debugging the reasons why a certain query holds, we can impose quantitative constrains on the traces and specify what kind of witness traces we wish to visualize. For traffic engineering purposes we may want to find a trace that has the lowest latency or the smallest number of hops. We may be interested in finding a trace that minimizes tunneling depth or the number of failed links required to execute a given trace, or we may wish to find a trace that balances several such measures simultaneously.

We shall start by defining atomic quantiative properties of network traces. Let $N = (V, E, s, t, L, \tau)$ be an MPLS network and let $F \subseteq E$ be the set of failed links. An *atomic quantity* is a function $p : ((E \setminus F) \times H)^* \to \mathbb{N}_0$ that for a given trace σ evaluates to a non-negative integer $p(\sigma)$ representing the quantitative measure of the trace. In our tool, we support the following atomic quantities of a network trace $\sigma = (e_1, h_1) \dots (e_n, h_n)$:

- $\mathit{Links}(\sigma) = n$ is the length of the trace,
- $\mathit{Hops}(\sigma) = |\{e \in \{e_1, \dots, e_n\} \mid s(e) \neq t(e)\}|$ is the number of hops, where we avoid counting links that are self-loops,

- $\mathit{Distance}(\sigma) = \sum_{i=1}^{n} d(e_i)$ is the distance for any distance function $d : E \to \mathbb{N}_0$, e.g., the geographical distance, latency or e.g. inverse bandwidth capacity,
- $\mathit{Failures}(\sigma) = \sum_{i=1}^{n-1} |\mathit{failed}(i)|$ where $\mathit{failed}(i) = \{e \mid (e, \omega) \in O_k,\ 1 \leq k < j\}$, where $\tau(e_i, \mathit{head}(h_i)) = O_1 O_2 \ldots O_m$, and where j, is the lowest index such that O_j is an active traffic engineering group, and
- $\mathit{Tunnels}(\sigma) = \sum_{i=1}^{n-1} \max(0,\ |h_{i+1}| - |h_i|)$ is the number of pushes of new MPLS labels on the existing label-stack.

The atomic quantity $\mathit{Failures}(\sigma)$ measures the minimal number of failed links which are necessary at each router in order to enable the feasibility of the trace σ. The function $\mathit{Tunnels}(\sigma)$ measures the positive increase in the label-stack height during the trace σ that corresponds to the number of tunnels created during the trace.

Consider again the traces for our running example from Figure 1c. We have $\mathit{Hops}(\sigma_0) = \mathit{Links}(\sigma_0) = 4$ and $\mathit{Hops}(\sigma_3) = \mathit{Links}(\sigma_3) = 5$. We also observe that $\mathit{Failures}(\sigma_2) = 1$ while $\mathit{Failures}(\sigma_3) = 0$. Finally, we can see that e.g. $\mathit{Tunnels}(\sigma_1) = 1$, $\mathit{Tunnels}(\sigma_2) = 2$ and $\mathit{Tunnels}(\sigma_3) = 0$.

We can now combine the atomic quantities in order to define composed criteria for trace weight specification, by constructing linear expressions of the form

$$\mathit{expr} ::= \ p \mid a \ * \ \mathit{expr} \mid \mathit{expr}_1 + \mathit{expr}_2$$

where p is an atomic quantity and $a \in \mathbb{N}$. A vector of linear expressions $(\mathit{expr}_1, \mathit{expr}_2, \ldots, \mathit{expr}_n)$ allows us to specify trace properties by priorities, so that expr_1 has a higher priority than expr_2 etc. For a trace σ, there is a natural evaluation of linear expressions to nonnegative integers and for a vector of linear expressions, we assume the lexicographical ordering $\sqsubseteq$ on vectors of nonnegative integers, by abuse of notation extended to traces.

Problem 2 (Minimum Witness Problem). For a network, a query that is satisfied in the network and a vector of linear expressions $(\mathit{expr}_1, \mathit{expr}_2, \ldots, \mathit{expr}_n)$, we want to find a witness trace σ such that $\sigma \sqsubseteq \sigma'$ for any other witness trace σ'.

Consider the query φ_4 in our running example from Figure 1 where we want to find witness traces that will minimize the vector $(\mathit{Hops}, \mathit{Failures}+3\cdot\mathit{Tunnels})$. The query has two witness traces σ_2 and σ_3 and when we evaluate them on the minimization vector, we get the pair $(5, 1 + 3 \cdot 2) = (5, 7)$ for σ_2 and $(5, 0 + 3 \cdot 0) = (5, 0)$ for σ_3. As lexicographically $(5, 0) \sqsubseteq (5, 7)$, the answer to the minimum witness problem is the trace σ_3. In general, there can be several minimum witness traces, and we may return any of those, or add another minimization criterion to the vector of linear expressions.

Fig. 2: Running example loaded in the AalWiNes GUI

4 Tool Implementation

We now give an overview of the tool architecture, its theoretical foundation and integration with the dataplane configuration. The front end of our tool provides a web-browser based visualization. The graphical interface allows us to load a number of predefined networks from the Internet Topology Zoo [33], the operator's network used in the experiments as well as the running example used in this tool paper. In the interface we can specify the query, including an online help for router names with interfaces as well as the sets of labels tested at each router. In options, we can set different parameters for the tool and graphically create the vector of linear expressions for the minimum trace specification. If a witness trace is discovered, the GUI visualizes the trace including the operations performed at each router. A screenshot in Figure 2 shows how to specify the minimization vector $(Hops, Failures + 3 \cdot Tunnels)$ and the corresponding witness trace. The GUI is written in JavaScript and the source code is available under the GPL3 license. The backend verification engine is running on a web server at `https://demo.aalwines.cs.aau.dk/` and there is also a packaged version of the tool that can be run locally without the use of a web server (and allows to input additional MPLS networks created by the user).

4.1 Verification methodology

Our tool is based on automata-theoretic approach that leverages a translation from the query satisfiability (in an MPLS network) to a reachability analysis of a pushdown automaton (with potentially infinitely many reachable configurations). As reachability in pushdown automata is decidable in polynomial time [34, 35], this approach has the potential of scaling to large networks.

The connection between MPLS networks and pushdown automata was first noticed in [7, 24] where the authors provide a command-line prototype implementation in Python with encouraging experiments showing the feasibility of the approach. They use a state-of-the-art pushdown model checker Moped [36, 37] for reachability checking on pushdown automata and show that they can verify complex network queries on network topologies with 20-30 routers in a matter of hours. However, the work in [24] is a purely qualitative approach and does not provide any support for quantitative analysis. In order to deal with quantitative aspects, we extend the approach from [24] and suggest a novel translation from the query satisfiability problem with minimization criteria for witness traces, into the framework of *weighted* pushdown automata [38]. The theoretical foundations for the verification of weighted pushdown automata have been developed in the area of dataflow analysis [39] where polynomial-time algorithms are known even for the weighted extension. The basic observation behind this automata-theoretic approach to reachability analysis of weighted pushdown automata is that the set of all reachable configurations in a pushdown system forms a regular language that can be effectively represented by a nondeterministic finite automaton (of polynomial size) with transitions annotated by weights. The length of the shortest path to reach a pushdown configuration then corresponds to the shortest accepting path in the finite automaton under that configuration. As the Moped tool employed in [24] cannot handle weighted pushdown automata, we develop a new weighted pushdown automata C++ library AalWiNes (available at `https://github.com/DEIS-Tools/AalWiNes`) to replace Moped. Our experiments show a significant (several orders of magnitude) speedup due to the novel translation method with optimized reduction methods as well as due to our efficient implementation of the backend engine.

4.2 Tool architecture

The details about the architecture of our tool are given in Figure 3. First we obtain a dataplane snapshot of the routing forwarding tables (including the failover rules) as described in Appendix A. If the network configuration changes, we need to obtain a new dataplane snapshot. The graphical user interface allows us to load the MPLS network, a query and possibly also a weight expression. The tool then constructs a pushdown automaton by

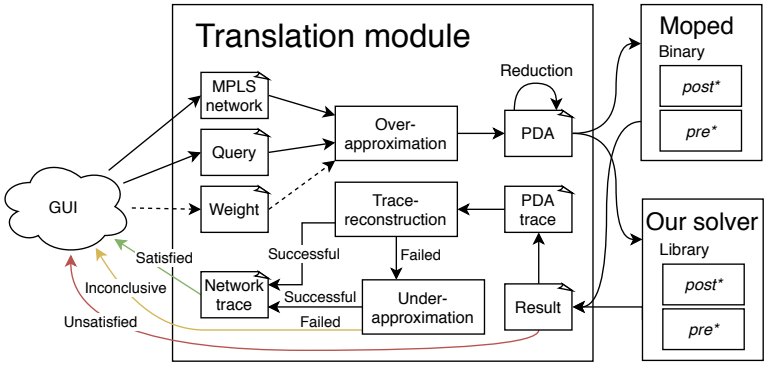


Fig. 3: Tool Architecture

means of over-approximation as the exact analysis requires to enumerate all of the (exponentially many) failure scenarios. Intuitively, over-approximation assumes that up to k links can fail at any router. This clearly includes all failure scenarios of up to k globally failed links but it may include additional traces that contain more than k failed links in total.

After this, the tool performs a series of reductions (based on static analysis that overapproximates the possible top-of-stack symbols in every given control state) on the constructed (weighted) pushdown automaton by removing redundant rules in order to decrease its size. The reduced pushdown is then sent either to the Moped engine (possible only if the weight requirements are not specified) or to our solver that accepts both weighted and unweighted pushdown automata. If the verification result says that the query is not satisfied, we achieve a conclusive answer and report it to the GUI. Otherwise, the produced network trace must be verified for its feasibility and the fact that it does not exceed in total k link failures (for a fixed trace this can be done in polynomial time). If the trace reconstruction succeeds, we have a witness trace (possibly one of the minimal ones in case the weight objective is given) and we can report that a query is satisfied. Otherwise, our tool constructs an under-approximating pushdown automaton where we add a global failed link counter and use this counter to guarantee that the total number of failed links is not exceeded. This produces only an under-approximation as in case of traces with loops, we may count the same failed link twice. If a valid trace is generated by the under-approximation, we can return it as a witness trace. Otherwise the answer is inconclusive. In our experiments on a real operator network, the answer was inconclusive for 8 out of 6,000 queries (0.13% of the total)—a more expensive analysis is then needed.

Query	Moped	Dual	Failures
$\langle$`smpls ip`$\rangle$ [$\cdot$ #$R6$] $\cdot^*$ [$\cdot$ #$R4$] $\langle$`smpls ip`$\rangle$ 1	9.57	0.82	41.23
$\langle$`smpls ip`$\rangle$ [$\cdot$ #$R2$] $\cdot^*$ [$\cdot$ #$R18$] $\langle$(`mpls`* `smpls`)? `ip`$\rangle$ 1	9.29	0.86	31.76
$\langle$`ip`$\rangle$ [$\cdot$ #$R0$] $\cdot^*$ [$\cdot$ #$R4$] $\langle$`ip`$\rangle$ 0	0.88	0.01	0.02
$\langle$[$449550] `ip`$\rangle$ [$\cdot$ #$R0$] $\cdot^*$ [$\cdot$ #$R5$] $\cdot^*$ [$\cdot$ #$R1$] $\langle$`ip`$\rangle$ 0	1.66	0.02	0.03
$\langle$[$449550] `ip`$\rangle$ [$\cdot$ #$R0$] $\cdot^*$ [$\cdot$ #$R5$] $\cdot^*$ [$\cdot$ #$R1$] $\langle$`ip`$\rangle$ 1	6.08	0.05	0.06
$\langle$`smpls? ip`$\rangle$ $\cdot^*$ $\langle\cdot$ `smpls ip`$\rangle$ 0	89.37	14.73	432.66

Table 1: Query verification time (in seconds)

5 Performance Evaluation

We evaluate the performance of our tool on a real-world network operator NORDUnet (`http://www.nordu.net/`) with 31 routers and more than 250.000 forwarding rules. The operator uses an advanced MPLS routing in its network, including numerous service labels by which it communicates with neighboring networks. In order to increase the variety of different types of networks, we create several variants of networks from Internet Topology Zoo [33] (having on average 84 routers and 240 routers at the largest instance) with label switching paths between any two edge routers and with local fast failover protection by introducing tunnels based on shortest paths; the queries are like in Table 1 and in our running example. The experiments were run on our cluster with AMD EPYC 7551 processors running at 2.55 GHz with boost disabled, using 16GB memory limit and 10 minutes timeout. A reproducibility artifact [22] includes the specific queries used in our experiments.

The operator asked us to verify a number of specific queries, including those in Table 1. The table shows the verification time for using Moped as the backend engine, our own engine (called Dual as it combines the over- and under-approximation approach) and our weighted verification engine that uses the *Failures* atomic quantity. Both the unweighted (Dual) and weighted (Failure) engine is a part of our AalWiNes verification suite. The results show that for three queries our weighted engine is on average about 4 times slower that Moped and for other three queries it is about 70 times faster than Moped. Our unweighted engine is always faster and has a 53 times speed up on average compared to Moped. The overhead of performing quantitative analysis is hence reasonable as the performance is comparable with the state-of-the art unweighted tool. Noticeably, the last query in the table takes significantly more verification time for all three engines. The reason is that its path constraint is very unspecific (allows for any sequence of routers) and the created pushdown system hence becomes significantly larger. We also note that [24] reports that the unweighted verification of similar queries on a network of comparable size

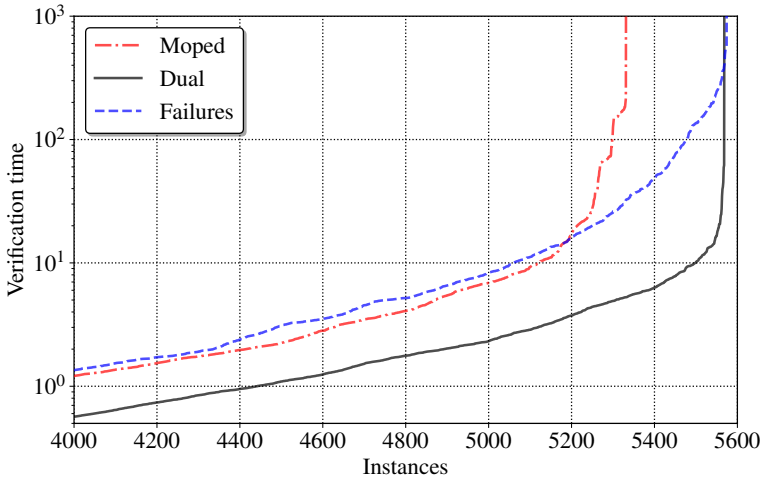


Fig. 4: Comparison on Topology Zoo networks

took between 28 minutes (for the simpler queries) and up to 109 minutes (for the more complex ones). This shows an improvement of several orders of magnitude and makes it possible to perform MPLS verification interactively for human operators, in particular in combination with our GUI ([24] is a command-line tool).

Finally, the plot in Figure 4 shows a comparison (note the logarithmic scale) of the verification times (in seconds) between Moped, our Dual unweighted approach and our weighted engine with the quantity *Failures* (we also run the experiment for the other quantitative measures and the verification times did not differ significantly). The plot includes over 5602 experiments on different queries on the networks from the Internet Topology Zoo database, ordered by their verification times. As the input format of all three engines is the same, all experiments were run with the same set of network topologies and the same queries. Again, we outperform Moped by almost an order of magnitude by using our unweighted engine. An interesting phenomenon can be observed for our weighted engine that behaves similarly as Moped on the smaller instances; however, on the difficult instances it is able to verify 6 more cases than our unweighted implementation. This is due to the fact that the guided search for shortest traces, that minimize the number of failures, allows us to find witness traces that are otherwise not discovered by the unweighted search; this highlights the benefits of quantitative analysis. This is further confirmed by the percentage of inconclusive answers that corresponds to 0.57% (32 inconclusive answers out of 5568) for the Dual unweighted approach and only 0.04% (2 inconclusive answers out of 5574) for the weighted engine optimizing the number of failures. In vast majority of cases we can hence use our

approximation approach that is guaranteed to run in polynomial time.

6 Conclusion

We presented an MPLS what-if analysis tool that not only provides an unprecedented performance in theory but also in practice, as demonstrated in our case study with a major network operator. We regard our contribution concerning the automated analysis of quantitative aspects as a first step, and believe that our paper opens interesting avenues for future research. We are currently improving the expressiveness of the query language.

Acknowledgements. We thank Henrik T. Jensen from NORDUnet for providing us with configuration data. The research is supported by DFF project QASNET and WWTF project ICT19-045.

References

[1] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian, "Leveraging SDN layering to systematically troubleshoot networks," in *Proc. ACM HotSDN*, 2013, pp. 37–42.

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

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

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

[5] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *Proc. ACM POPL*, 2014, pp. 113–126.

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012, pp. 113–126.

[7] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating MPLS networks," in *Proc. IEEE INFOCOM*, 2018, pp. 1799–1807.

[8] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. USENIX NSDI*, 2018, pp. 579–594.

[9] ——, "Network-wide configuration synthesis," in *CAV'17*, ser. LNCS, vol. 10427. Springer, 2017, pp. 261–281.

[10] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, 2011, pp. 290–301.

[11] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. ACM SIGCOMM*, 2017, pp. 155–168.

[12] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. USENIX NSDI*, 2013, pp. 15–27.

[13] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. USENIX NSDI*, 2014, pp. 87–99.

[14] M. Chiesa, A. Kamisiński, J. Rak, G. Rétvári, and S. Schmid, "Fast recovery mechanisms in the data plane," May 2020.

[15] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *Proc. USENIX NSDI*, 2013, pp. 113–126.

[16] M. Chiesa, I. Nikolaevskiy, S. Mitrović, A. Gurtov, A. Madry, M. Schapira, and S. Shenker, "On the resiliency of static forwarding tables," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1133–1146, 2016.

[17] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," RFC 4090, May 2005.

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

[19] A. Atlas and A. D. Zinin, "Basic specification for IP fast reroute: Loop-free alternates," RFC 5286, Sep. 2008.

[20] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "IP fast rerouting for multi-link failures," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 3014–3025, 2016.

[21] S. Smith, *Introduction to MPLS*, https://www.cisco.com/c/dam/global/fr_ca/training-events/pdfs/Intro_to_mpls.pdf, 2003, visited: 19/05/2020.

[22] P. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. Schrenk, and J. Srba, "Artifact for "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks"," Zenodo, Oct. 2020.

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

[24] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[25] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. USENIX NSDI*, 2015, pp. 469–483.

[26] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM*, 2016, pp. 300–313.

[27] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. USENIX OSDI*, 2016, pp. 217–232.

[28] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *Proc. USENIX NSDI*, 2020, pp. 953–967.

[29] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *Proc. USENIX NSDI*, 2020, pp. 201–219.

[30] A. Shukla, S. J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann, "Toward consistent SDNs: A case for network state fuzzing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 668–681, 2020.

[31] K. G. Larsen, S. Schmid, and B. Xue, "WNetKAT: A weighted SDN programming and verification language," in *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, ser. LIPIcs, vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 18:1–18:18.

[32] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with wise," in *Proc. ACM SIGCOMM*, 2008, pp. 99–110.

[33] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[34] J. R. Büchi, "Regular canonical systems," *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[35] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[36] S. Schwoon, "Moped," in *http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/*, 2002.

[37] ——, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, 2002.

[38] W. Kuich and A. Salomaa, Eds., *Semirings, Automata, Languages*. Berlin, Heidelberg: Springer-Verlag, 1985.

[39] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Science of Computer Programming*, vol. 58, no. 1-2, pp. 206–263, 2005.

A Appendix

By default, our tool accepts a generic and vendor agnostic XML input format for a network. The input is split into a *topology* definition and a *routing* definition and examples are given below.

topo.xml

```
<network>
  <routers>
    <router name="R0">
      <interfaces>
        <interface name="ae1.11"/>
        <interface name="ae5.0"/>
        ...
      </interfaces>
    </router>
    ...
  </routers>
  <links>
```

```
    <sides>
      <shared_interface interface="et-3/0/0.2" router="R0"/>
      <shared_interface interface="et-1/3/0.2" router="R3"/>
    </sides>
  </links>
  ...
</network>
```

route.xml

```
<routes>
  <routings>
    <routing for="R0">
      <destinations>
        <destination from="ae1.11" label="$300292">
            <te-group>
              <routes>
                <route to="et-1/1/0.0">
                  <actions>
                    <action arg="$300050" type="swap"/>
                    <action arg="496505" type="push"/>
                  </actions>
                </route>
              </routes>
            </te-group>
            ...
        </destination>
        ...
      </destinations>
    </routing>
    ...
  </routings>
</routing>
```

A.1 IS-IS input

Our tool accepts topological description and routing tables exported directly from an IS-IS system; to do so we run the following commands on each router in the network:

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

To correctly reconstruct the network configuration, an additional mapping file has to be constructed. Each line in the mapping file corresponds to a single logical routing entity and is given in the form `<aliases>:<adj.xml>:<route-ft.xml>:<pfe.xml>`. Edge routers can also be defined by omitting the xml-files. In the case of edge routers, the routing-table is assumed empty, and such routers will act as sink-nodes in the network. An example of such a mapping file is given below.

```
192.0.0.1,R1:R1-adj.xml:R1-route.xml:R1-pfe.xml
192.0.0.2,10.10.0.2,E1
...
```

A network given as an extract from an IS-IS system can be turned into the vendor agnostic format by calling directly our binary and providing the `-write-topology topo.xml` and `-write-routing route.xml` options.

A.2 Location data

To correctly visualize the network in the GUI, an additional location mapping has to be provided giving latitude and longitude to each router. This information is also used for computing the physical distance between routers used in the minimum trace specification. An example is given below.

```
{ "R0": { "lat": 46.5,"lng": 7.3}, ...  }
```

Paper E

PDAAAL: A Library for Reachability Analysis of Weighted Pushdown Systems

Peter Gjøl Jensen, Stefan Schmid,
Morten Konggaard Schou, and Jiří Srba

The paper has been published in:
Automated Technology for Verification and Analysis (ATVA 2022),
Lecture Notes in Computer Science, vol. 13505, pp. 225-230, Springer, 2022.
https://doi.org/10.1007/978-3-031-19992-9_14

The layout has been revised.

Abstract

We present PDAAAL, an open source C++ library and tool for weighted reachability analysis of pushdown systems, including generation of both shortest and longest witness traces. We consider totally ordered weight domains which have important applications, e.g. in network verification, and achieve a speedup of two orders of magnitude compared to the state-of-the-art tool WALi. Our tool further extends the state of the art by supporting the generation of the longest trace in case it exists, or reporting that no finite longest trace can be generated. PDAAAL is provided both as a stand-alone tool accepting JSON files and as a C++ library. This allows for integration in software pipelines as well as in verification tools like AalWiNes.

1 Introduction

Pushdown automata are a fundamental model in computer science and they are often used as an underlying formalism for data-flow analysis of recursive programs [1–4], parsing of XML streams [5], modelling of network protocols [6, 7] and others [8, 9]. The verification questions on different types of models can be reduced to reachability analysis for pushdown systems.

In order to support quantitative extensions of such systems, we need to study weighted extensions of pushdown automata. In general, these weights are defined over an idempotent semiring and we need to consider meet-over-all-paths values for reaching certain pushdown configurations. There is a rich literature of theory showing the application of weighted pushdown automata [4, 8–10] to various application domains and several tools for the reachability analysis of pushdown systems exist, including the tools Moped [11] used for the analysis of Java programs (in the jMoped framework [3]), WPDS++ [12] for program analysis as well as its more recent successor WALi [13] employed in tools like ICRA [9] performing interprocedural compositional recurrence analysis and the static analysis tool Phasar [4] for C/C++ programs.

We present an open source C++ library and stand-alone tool PDAAAL for efficient reachability analysis of pushdown automata over the weight domain of totally ordered idempotent semirings. The study of such totally ordered semirings is fundamental and has important applications, e.g., in the context of the verification of communication networks [6, 7]. PDAAAL implements the classical pre^* and $post^*$ saturation algorithms for unweighted pushdown systems, including the new $dual^*$ algorithms [14] and extends these algorithms for weighted reachability analysis, computing the weights of not only the shortest traces but also the longest traces, while returning such trace witnesses in case they exist. The study of longest traces is practically relevant, as it allows, for example, to perform a worst-case analysis of the routing paths in a communication network, e.g., in terms of delay or size of packet headers [7]. It

is, however, also challenging to analyze, as the longest trace may be unbounded and hence impossible to compute directly. To the best of our knowledge, PDAAAL is the first tool providing the exact computation of the longest traces as the debugging information.

We introduce the formalism of weighted pushdown systems (Section 2), present the implemented algorithms and tool usage (Section 3) and compare the performance of PDAAAL with the state-of-the-art tool WALi for weighted reachability analysis (Section 4) where we observe up to two orders of magnitude faster performance. Finally, we elaborate on a specific use case in network verification (Section 5) related to MPLS networks [7].

2 Weighted Pushdown Systems and Reachability

PDAAAL can accept weights from the domain of totally ordered idempotent semirings $S = (D, \sqcap, \oplus, \top, \bot)$. An example of a weight domain for computing the shortest paths is $S_1 = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ where weights are natural numbers including infinity, the weights are additive along a single path and minimum is the meet-over-all-path operation. A domain for the computation of the longest path is $S_2 = (\mathbb{Z} \cup \{-\infty\}, \max, +, -\infty, 0)$.

Definition 1. A Weighted Pushdown System (WPDS) over a weight semiring $S = (D, \sqcap, \oplus, \top, \bot)$ is a tuple (P, Γ, Δ) where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, and the set of *rules* Δ is a finite subset of $(P \times \Gamma) \times D \times (P \times \Gamma^*)$, written $\langle p, \gamma \rangle \overset{d}{\hookrightarrow} \langle p', w \rangle$, if $((p, \gamma), d, (p', w)) \in \Delta$.

A *configuration* in a pushdown system is a pair $\langle p, w \rangle$ where $p \in P$ is the current control location and $w \in \Gamma^*$ is the stack content (head of the stack on the left). A WPDS induces a labelled transition system $T = (P \times \Gamma^*, D, \Rightarrow)$, where for all $w' \in \Gamma^*$ $\langle p, \gamma w' \rangle \overset{d}{\Rightarrow} \langle p', w w' \rangle$, provided that there is a pushdown rule $\langle p, \gamma \rangle \overset{d}{\hookrightarrow} \langle p', w \rangle$. We write $c_0 \overset{d}{\Rightarrow}^{\oplus} c_n$ if there is a path in the labelled transition system $c_0 \overset{d_1}{\Longrightarrow} \ldots \overset{d_n}{\Longrightarrow} c_n$ such that $d = d_1 \oplus \ldots \oplus d_n$. The *distance* between two configurations c and c' is given by $\delta(c, c') = \bigsqcap \{d \mid c \overset{d}{\Rightarrow}^{\oplus} c'\}$. If S is a bounded idempotent semiring (i.e. has no infinite descending chains), the distance is well defined. If S is unbounded, the supremum may not be in the domain D; for example in the semiring S_2, the distance is ∞ if there is a positive-weight loop.

The problem solved by PDAAAL is: given a WPDS (P, Γ, Δ) and two regular sets of configurations $C, C' \subseteq P \times \Gamma^*$, compute the distance $\bigsqcap \{\delta(c, c') \mid c \in C, c' \in C'\}$ and return a witness trace (if any) with this distance.

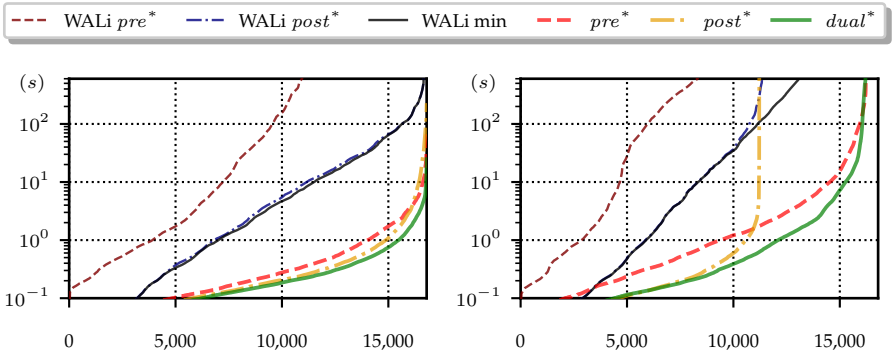
3 Implemented Algorithms and PDAAAL Architecture

It is well known that the sets of all predecessors $pre^*(C)$ and successors $post^*(C)$ of a regular set of pushdown configurations C are also regular [15]. The classical pre^* and $post^*$ saturation algorithms [2, 16, 17] solve reachability for pushdown systems without weights. Schwoon [17] describes how to find a shortest witness trace for totally ordered weight domains by using a priority queue to select the next step of the saturation. This is later generalized to bounded idempotent semirings [8], and implemented in the tool WALi [13]. Here the saturation algorithms use a workset where transitions may be added multiple times, hence possibly loosing some efficiency compared to the priority queue that exploits the total ordering. Extensions to unbounded semirings are considered in [10] by detecting that exceeding a given number of iterations of the saturation algorithm causes nontermination of the procedure. PDAAAL implements the ideas from [10] to pre^*, $post^*$ as well as $dual^*$ (combination of the first two approaches) for unbounded but totally ordered weight domains.

To achieve a high performance, we employ numerous algorithmic optimizations. We extend the early termination and bidirectional-search ($dual^*$) technique from unweighted pushdowns [14] to shortest trace queries for weighted systems. The main challenge here is that the on-the-fly construction of the product automata must keep track of the weight of the best path to any state, and the saturation only terminates if the best weight of an accepting path is no higher than the current weight in the priority queue in the saturation. For longest trace queries the $dual^*$ approach simply interleaves the saturation of pre^* and $post^*$ and returns when either of them terminates. We also efficiently handle rules that apply to any top-of-stack label, using of a wildcard flag in the precondition, and adapting the pre^* and $post^*$ algorithms to efficiently handle wildcards.

PDAAAL is designed to be included as a library in other C++ projects, but it also functions as a stand-alone tool with JSON parsers for pushdown systems and $\mathcal{P}$-automata (nondeterministic automata used to represent regular sets of pushdown configurations). The tool has predefined weight domains for integers and natural numbers as well as vectors of these. In all cases, the weight semiring can either minimize or maximize the weight, depending on whether a shortest or longest trace is required. Other weight domains can be defined by the user, when PDAAAL is used as a C++ library.

As an example, a $\mathcal{P}$-automaton for the following set of pushdown configurations $\{\langle p_0, BA\rangle, \langle p_0, A\rangle, \langle p_1, A\rangle\}$ can be defined either in JSON format, or by using regular expressions for the stack, symbols '<' and '>' to denote configurations, and the symbol '|' to union multiple configuration sets: `< [p0], [B]? [A] > | < [p1], [A] >`.



(a) Shortest latency **(b)** Longest latency

Fig. 1: Performance plots of WALi (pre^*,$post^*$ and minimum of both) in thin lines, compared to PDAAAL (pre^*, $post^*$ and $dual^*$) in thick lines; all instances on x-axis are independently sorted by the increasing verification time that is plotted on y-axis (log-scale) in seconds.

To run PDAAAL from the command line, an input file must be provided along with the algorithm to use: `-e 1` ($post^*$), `-e 2` (pre^*), or `-e 3` ($dual^*$) and the trace type `-t 0` (no trace), `-t 1` (any trace), `-t 2` (shortest trace), or `-t 3` (longest trace). For instance to run the $post^*$ shortest trace algorithm: `pdaaal --input example.json -e 1 -t 2`.

4 Comparison with State-of-the-Art

The first library for weighted pushdown systems, called WPDS [17], was provided by Schwoon and used in Moped version 2. Later, WPDS++ [12] was developed by Reps et al. and included further performance optimizations. The state-of-the-art tool WALi [13] was developed as a successor of WPDS++ and it is used as a backend in recent static analyzers ICRA [9] and Phasar [4] .

We compare PDAAAL to WALi by running the shortest and longest trace queries on weighted pushdown systems produced by AalWiNes [7] on a large benchmark of real communication networks from ISP providers. All together, we run 16,800 reachability queries on pushdown systems of varying sizes. WALi does not support a generation of the longest traces, unless a bound on the weight of the longest trace is known a priori. In order to enable this, we set the bound to the highest possible value of 32bit integer. On contrary, the implementation in PDAAAL is able to effectively compute a bound on the number of iterations, and hence it guarantees the termination even for unbounded longest traces.

Figure 1 shows the results comparing WALi and PDAAAL. We consider

both the computation of shortest traces and longest traces where the weight domain represents the latency (which is additive along a pushdown trace). PDAAAL supports both pre^*, $post^*$ and $dual^*$ (interleaving of pre^* and $post^*$), while WALi does not support $dual^*$. We instead present the minimum of the verification time of pre^* and $post^*$, which shows an improvement on the largest instances for the longest latency. For the shortest trace experiment, all variants of PDAAAL saturation algorithms outperform WALi by several orders of magnitude. For the longest traces, this is also the case for our $dual^*$ algorithm, even though the $post^*$ algorithm times out about at the same instance as WALi. We can also observe that our pre^* implementation is in general performing as good (or even better) than our $post^*$, while this is not the case for WALi.

PDAAAL is available on `https://github.com/DEIS-Tools/PDAAAL` together with specifications of input/output formats and how to run the tool. A reproducibility package is available at `https://doi.org/10.5281/zenodo.6833493`.

5 Applications

Pushdown automata find broad and practical applications in many domains where verification tasks are often reduced to a pushdown reachability analysis. As an example, PDAAAL can be used to model MPLS networks, a popular and widely-used type of communication network used by most Internet Service Providers for efficient traffic engineering [7]. MPLS networks interconnect a set of routers which forward packets, where packets contain stacks of labels which can be pushed and popped, and the forwarding is based on the top-of-stack label. Such networks can hence be modelled as pushdown systems.

PDAAAL can be used in combination with AalWiNes [7] as part of a what-if analysis tool (behavior under link failures) to ensure a dependable service and policy-compliant routing. In particular, PDAAAL's support for longest traces is attractive to perform a worst-case analysis of the network's routing behavior. For example, PDAAAL can be used to compute the longest possible routes that may occur under one or multiple link failures, both in terms of the number of hops (which is directly related to the amount of bandwidth resources consumed in the network) as well as in terms of the overall delay (an important metric for latency-critical applications). Furthermore, PDAAAL can also be used to verify further quantitative metrics of interest. An online demo is available at `http://demo.aalwines.cs.aau.dk`.

Similar applications for the longest trace analysis also arise in other domains, allowing to perform worst-case time analyses of possible control flows in recursive programs or the execution of parsers of XML streams, shedding light on the possible overheads of such operations.

6 Conclusion

We presented PDAAAL, a tool for reachability analysis of weighted pushdown automata over possibly unbounded weight domains. Our tool can be used also as a library, and it is integrated into a recent network analysis tool AalWiNes that relies on pushdown systems produced from widely used MPLS networks. Apart from being two orders of magnitude faster than the state-of-the-art competitor, it supports the detection of the existence of longest traces which finds practical applications in e.g., the analysis of network protocols. Our tool uses unbounded but totally ordered weight domains but despite of this limitation, it finds numerous applications and can in the case of totally ordered domains replace the backend weighted engines like Moped, WPDS++ and WALi with a generic, modern and efficient library.

References

[1] J. Esparza and J. Knoop, "An automata-theoretic approach to interprocedural data-flow analysis," in *FOSSACS'99*, ser. LNCS, vol. 1578. Springer, 1999, pp. 14–30.

[2] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[3] D. Suwimonteerabuth, S. Schwoon, and J. Esparza, "jMoped: A java bytecode checker based on Moped," in *TACAS'05*, ser. LNCS, vol. 3440. Springer, 2005, pp. 541–545.

[4] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An inter-procedural static analysis framework for C/C++," in *TACAS'19*, ser. LNCS, vol. 11428. Springer, 2019, pp. 393–410.

[5] V. Kumar, P. Madhusudan, and M. Viswanathan, "Visibly pushdown automata for streaming XML," in *WWW'07*. ACM, 2007, pp. 1053–1062.

[6] S. Schmid and J. Srba, "Polynomial-time what-if analysis for prefix-manipulating MPLS networks," in *Proc. IEEE INFOCOM*, 2018, pp. 1799–1807.

[7] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[8] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Science of Computer Programming*, vol. 58, no. 1-2, pp. 206–263, 2005.

[9] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps, "Compositional recurrence analysis revisited," in *Proc. ACM PLDI*, 2017, pp. 248–262.

[10] M. Kühnrich, S. Schwoon, J. Srba, and S. Kiefer, "Interprocedural dataflow analysis over weight domains with infinite descending chains," in *FOSSACS'09*, ser. LNCS, vol. 5504. Springer-Verlag, 2009, pp. 440–455.

[11] S. Schwoon, "Moped," in *http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/*, 2002.

[12] N. Kidd, T. Reps, D. Melski, and A. Lal, "WPDS++: A C++ library for weighted pushdown systems," *Univ. of Wisconsin*, 2004.

[13] N. Kidd, A. Lal, and T. Reps, "WALi: The weighted automaton library," 2007. [Online]. Available: https://research.cs.wisc.edu/wpis/wpds/wali/

[14] P. G. Jensen, S. Schmid, M. K. Schou, J. Srba, J. Vanerio, and I. van Duijn, "Faster pushdown reachability analysis with applications in network verification," in *Automated Technology for Verification and Analysis (ATVA 2021)*, ser. LNCS, vol. 12971. Springer, 2021, pp. 170–186.

[15] J. R. Büchi, "Regular canonical systems," *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[16] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," in *INFINITY'97*, ser. ENTCS, vol. 9. Elsevier, 1997, pp. 27–37.

[17] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, 2002.

Paper F

Differential Testing of Pushdown Reachability with a Formally Verified Oracle

Anders Schlichtkrull, Morten Konggaard Schou,
Jiří Srba, and Dmitriy Traytel

The paper has been published in:
Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD 2022), pp. 369-379, TU Wien Academic Press, 2022.
https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_44

The layout has been revised.

Abstract

Pushdown automata are an essential model of recursive computation. In model checking and static analysis, numerous problems can be reduced to reachability questions about pushdown automata and several efficient libraries implement automata-theoretic algorithms for answering these questions. These libraries are often used as core components in other tools, and therefore it is instrumental that the used algorithms and their implementations are correct. We present a method that significantly increases the trust in the answers provided by the libraries for pushdown reachability by (i) formally verifying the correctness of the used algorithms using the Isabelle/HOL proof assistant, (ii) extracting executable programs from the formalization, (iii) implementing a framework for the differential testing of library implementations with the verified extracted algorithms as oracles, and (iv) automatically minimizing counter-examples from the differential testing based on the delta-debugging methodology. We instantiate our method to the concrete case of PDAAAL, a state-of-the-art library for pushdown reachability. Thereby, we discover and resolve several nontrivial errors in PDAAAL.

1 Introduction

In 1964, Büchi [1] proved that the possibly infinite set of all reachable pushdown configurations (from a given initial configuration) can be effectively described by a regular language. In fact, even for a given regular set of pushdown configurations, its $post^*$ and pre^* closures (representing all forward and backward reachable configurations from a given set of configurations) are also regular. Büchi's *automata-theoretic approach* gave rise to a rich theory of pushdown reachability with numerous algorithms and applications to, e.g., interprocedural control-flow analysis of recursive programs [2, 3], model checking [4–7], communication network analysis [8–10] and others. A number of tools have been developed to support the theory, including Moped [5, 6], WALi [11], and PDAAAL [12] with applications ranging from the static analysis of Java [6] and C/C++ code [13, 14] to the analysis of MPLS communication protocols [9].

Even though the automata-theoretic approach for pushdown reachability is based on relatively simple saturation procedures, the proofs of correctness are nontrivial and the implementation of the algorithms in the different tools often includes numerous performance optimizations as well as additional improvements to the theory itself [12]. To be able to rely on the output of model checking tools and other applications of pushdown reachability, it is important that the theory is not only sound but also correctly implemented. A positive reachability answer is typically accompanied by a finite evidence (trace) that can function as an efficiently checkable certificate. A negative answer is, on the other hand, much harder to check, and designing a finite evidence for non-reachability is difficult, primarily because the number of reachable push-

down configurations can be infinite. One approach is to establish an invariant that (i) includes the initial configuration(s) of the system, (ii) is maintained by the transition relation and (iii) has an empty intersection with the set of undesirable configurations. Such approaches have been studied [15, 16] but are usually incomplete and require another complex tool (that can be error-prone, too) to verify such invariants.

We instead use a proof assistant, Isabelle/HOL [17] (§2), to formally verify the correctness of the pushdown reachability algorithms $post^*$ (forward search), pre^* (backward search), and $dual^*$ (bi-directional search) (§3) that lie at the heart of the automata-theoretic analysis of pushdown systems [7, 12, 18]. From the formalization of pre^*, we extract an executable program with strong correctness guarantees (§4). For a given input, the extracted program's output can be compared with the output of other, unverified but optimized tools solving the same problem (§5). This approach is known as differential testing [19–21] with a twist that the testing oracle has been formally verified and thus is extremely trustworthy. When testing reveals a disagreement between a verified and an unverified algorithm, we know who is to blame. To help localize errors in unverified algorithms, we minimize the tests causing disagreement using the delta-debugging technique [22]. Our main contributions are as follows.

- The formalization of $post^*$, pre^* and $dual^*$ algorithms in Isabelle/HOL and verification of their correctness based on the proofs provided by Schwoon [18] for $post^*$ and pre^*, and following Jensen et al. [12] for $dual^*$.
- The refinement to and the extraction of an executable program of the formalized pre^* algorithm that serves as the verified oracle for differential testing.
- The automatic minimization of the input automata in cases where an unverified tool disagrees with the oracle.
- The application of our method to a modern state-of-the-art library for pushdown reachability, PDAAAL [12], and the identification, localization (using the minimized counter-examples), and correction of three, previously unknown, implementation errors (§6). The corrected implementation passes all differential tests successfully.

Our Isabelle formalization as well as the case study are publicly available [23].

Related work Differential testing with a verified oracle has been used in the context of runtime verification and automatic theorem proving. The runtime monitor VeriMon [24, 25] served as the verified oracle used to detect errors in unverified monitors. Compared to our approach, VeriMon's differential testing case study is from a different application domain, does not include

exhaustive test generation for small input sizes (which is difficult in runtime monitoring) and does not minimize the tests automatically. To assess its performance but also to evaluate the benchmark's correctness, the verified first-order prover RPx [26] was evaluated on a standard benchmark for first-order logic problems. RPx's answers have in all cases coincided with the expected ones recorded in the benchmark.

The verified C compiler CompCert [27] and several verified distributed systems [28–30] have been themselves put onto the testbed [31, 32]. A few errors in these tools' unverified parts or in scenarios violating the verification assumptions were found, but none in the verified components themselves.

Many works extract efficient executable code from formalizations, but do not use it as an oracle in testing. Examples include verified model checkers for LTL [33] and timed automata [34] and verified algorithms for finite automata [35–38] and context-free grammars [39, 40].

The only formalization of pushdown automata we are aware of is part of Lammich et al.'s work on dynamic pushdown networks (DPN) [41]. Lammich describes the Isabelle formalization of an executable pre^* algorithm for DPNs stemming from this work in an unpublished technical report [42]. DPNs generalize pushdown automata, but their $post^*$ is not regular [43] and so we cannot extend this work for our purposes. Moreover, Lammich's formalization does not support ε-transitions in the underlying automata, an essential component needed for our formalization of $post^*$ and $dual^*$.

Background definitions Let P be a finite set of control locations and Γ a finite stack alphabet. A *pushdown system* (*PDS*) is a tuple (P, Γ, Δ), where $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *rules*, written $(p, \gamma) \hookrightarrow (q, w)$ whenever $((p, \gamma), (q, w)) \in \Delta$. Without loss of generality, we assume $|w| \leq 2$, so that $w = \varepsilon$ represents a *pop* operation that removes the topmost stack symbol, $|w| = 1$ is a *swap* that replaces the topmost symbol with another one, and $|w| = 2$ is a *push* that incorporates a *swap* followed by adding a new symbol on top.

A *configuration* of a pushdown system is a pair (p, w) of the current control location $p \in P$ and the current stack content $w \in \Gamma^*$ where we assume that the top of the stack is on the left. The set of all configurations is denoted by $\mathcal{C}$. A PDS can take a *computation step* $(p, \gamma w') \Rightarrow (q, ww')$ between configurations whenever $(p, \gamma) \hookrightarrow (q, w)$ and $w' \in \Gamma^*$. For a given $C \subseteq \mathcal{C}$, we define $post^*(C) = \{c' \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c \in C\}$ and $pre^*(C) = \{c \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c' \in C\}$.

The *reachability problem* for PDSs is to decide whether $c \Rightarrow^* c'$ for configurations c and c', and it is equivalent to asking whether $c' \in post^*(\{c\})$ or equivalently whether $c \in pre^*(\{c'\})$. Büchi [1] showed that for any regular set $C \subseteq \mathcal{C}$, the sets $post^*(C)$ and $pre^*(C)$ are also regular.

To represent regular sets of pushdown configurations, we use *P*-

automata [18], which are nondeterministic finite automata with multiple initial states for each of the control locations from the set P. Formally, let N be a finite set of noninitial states and $F \subseteq P \cup N$ a finite set of final states. A *P-automaton* is a tuple $\mathcal{A} = (P \cup N, \rightarrow, P, F)$ with the transition relation $\rightarrow \subseteq (P \cup N) \times \Gamma \times (P \cup N)$ so that $P \cup N$ is the set of its states and the pushdown alphabet Γ is the input alphabet of the automaton. The language $L(\mathcal{A})$ of P-automaton $\mathcal{A}$ contains the pushdown configurations accepted by $\mathcal{A}$: a configuration $(p, w) \in P \times \Gamma^*$ is accepted if and only if there is a path from p to q for some $q \in F$ in the P-automaton (defined via the transition relation $\rightarrow$) labelled with w. The *reachability problem for P-automata* is as follows: given a PDS (P, Γ, Δ) and P-automata $\mathcal{A}_1$ and $\mathcal{A}_2$, does there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ?

2 Isabelle/HOL

Isabelle/HOL [17] is a proof assistant based on classical higher-order logic (HOL), a simply typed lambda calculus with Hilbert choice, axiom of infinity, and rank-1 polymorphism. We present our formalization using HOL's syntax, which mixes functional programming and mathematical notation.

Types are built from type variables $'a$, $'b$, ... and type constructors like pairs _ × _ and functions _ ⇒ _ (both written infix) and sets _ *set* (written postfix). Type constructors can also be nullary, e.g., the Boolean type *bool*. Type variables can be restricted by type classes: $'a :: \mathit{finite}$ is a type variable $'a$ that can only be instantiated with finite types (i.e., types with finitely many inhabitants). New type constructors are introduced as abbreviations for complex type expressions and as inductive datatypes using commands **type_synonym** and **datatype** respectively, e.g., the types of transitions **type_synonym** $('state, 'label)\ transition = 'state \times 'label \times 'state$ and finite lists **datatype** $'a\ list = [] \mid 'a \# ('a\ list)$.

Terms are built from variables x, y, ..., constants c, d, ..., lambda abstractions $\lambda x.\ t$ and applications written as juxtaposition $f\ x$. Isabelle includes many constants and syntax for them, e.g., infix operators ∧, ∨, ⟶, ⟷, ∈, ∪, ∩, unbounded and bounded quantifiers $\exists x.\ P\ x$ and $\forall y \in A.\ Q\ y$, and set comprehensions $\{x.\ P\ x\}$. Non-recursive functions are defined and given readable syntax using the **definition** command:

definition image (**infix** ‘) **where**
  f ‘ $A = \{y.\ \exists x \in A.\ y = f\ x\}$

Type annotations like image :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ can be omitted as they are inferred. Recursive definitions are supported using the **fun** command:

fun append (**infix** @) **where**
  $[] @ ys = ys \mid (x \# xs) @ ys = x \# (xs @ ys)$

```
locale LTS = fixes trans_rel :: ('state, 'label) transition set begin
  definition step_relp (infix ⇒) where
    c ⇒ c' ⟷ (∃l. (c, l, c') ∈ trans_rel)
  definition step_starp (infix ⇒*) where
    c ⇒* c' ⟷ step_relp** c c'
  definition pre_star C = {c'. ∃c ∈ C. c' ⇒* c}
  definition post_star C = {c'. ∃c ∈ C. c ⇒* c'}
  definition srcs = {p. ∄q γ. (q, γ, p) ∈ trans_rel}
  definition sinks = {p. ∄q γ. (p, γ, q) ∈ trans_rel}
  inductive_set trans_star where
    (p, [], p) ∈ trans_star
  | (p, γ, q') ∈ trans_rel ⟶ (q', w, q) ∈ trans_star ⟶
    (p, γ#w, q) ∈ trans_star
end
```

Fig. 1: The locale for labeled transition systems

Internally, **fun** performs an automatic termination proof. More complex recursion schemes may require a manual proof.

Another way to define a function is as Prolog-style monotone rules. The **inductive** command allows such definitions as least fixed points. Take, e.g., the reflexive transitive closure:

```
inductive rtranclp (_**) where
  R** x x | R x y ⟶ R** y z ⟶ R** x z
```

Theorems and lemmas are terms of type *bool* that have been proved to be equivalent to True. All proofs pass through Isabelle's kernel, which relies only on a few well-understood reasoning rules such as modus ponens. We refer to a textbook [44] for a practical introduction to proving in Isabelle.

Structures and assumptions common to many theorems can be organized via locales [45]—Isabelle's module mechanism for fixing parameters and stating and assuming their properties. In the context of a locale, the parameters are available as constants and the assumptions as facts. Locales can be interpreted, which involves instantiating the parameters and proving the assumptions. As the result, one obtains the (instantiated) theorems proved in the context of the locale.

Consider our locale for labeled transition systems (LTSs) in Fig. 1. It fixes the parameter trans_rel, and its context consists of the definitions between the **begin** and **end** keywords. All definitions should be self-explanatory except perhaps trans_star: the set of triples (p, w, q) for which the LTS can move from p to q by consuming word w. This relation is defined inductively, first for the empty sequence and then extending it by one more symbol—here we use in

conjunction two assumptions on the symbol γ and sequence w. (Following an Isabelle convention, we formalize it equivalently as two implications.) In the formalization, the locale has more definitions than shown here and a number of lemmas. Outside LTS's context, we can access its definitions, e.g., pre_star is available under the name LTS.pre_star and can be applied to any transition relation A and a set of states C as follows: LTS.pre_star A C.

3 Pushdown Reachability

We formalize pushdown systems (PDSs) and saturation algorithms for calculating pre^* and $post^*$ following Schwoon [18] and $dual^*$ following Jensen et al. [12].

Fig. 2 shows our modeling of PDSs. We use type variables to represent control locations ($'ctr\_loc$) and stack labels ($'label$). We introduce types for operations ($'label\ op$), rules ($('ctr\_loc, 'label)\ rule$) and configurations ($('ctr\_loc, 'label)\ conf$). A PDS is given by the locale PDS, which fixes a set of rules Δ. Each PDS gives rise to an unlabeled transition relation, which we model by an LTS step with label ()—the only element of type $unit$. The definition is a non-recursive **inductive** definition. We use the **interpretation** command to interpret LTS with step. This means that pre_star refers to LTS.pre_star step in PDS. Likewise, trans_star refers to LTS.trans_star step and similarly for other LTS definitions. The type $('ctr\_loc, 'noninit)\ state$ represents P-automata states, where $'noninit$ is the type variable for noninitial states. The locale PDS_with_finals extends PDS with a set of final initial states F_inits and final noninitial states F_noninits. For the rest of this section, we work within the PDS_with_finals locale. In this locale, a P-automaton is a set of transitions.

3.1 Nondeterministic pre^* Saturation

Schwoon [18] presents the pre^* saturation which is a nondeterministic algorithm that given a P-automaton A returns a P-automaton whose language is pre_star (lang A). The algorithm proceeds by iteratively adding transitions to A. In each step, the algorithm nondeterministically chooses an transition to add that satisfies a number of criteria. The P-automaton is saturated when no more transitions can be added. We formalize a step of the algorithm by the relation:

inductive pre_star_rule **where**
(Init $p, \gamma, q) \notin A \longrightarrow (p, \gamma) \hookrightarrow (p', w) \longrightarrow$
(Init p', lbl $w, q) \in$ LTS.trans_star $A \longrightarrow$
pre_star_rule A $(A \cup \{(\text{Init } p, \gamma, q)\})$

The pre_star_rule relation relates two P-automata if the latter can be obtained from the former via one step of the algorithm. The criteria of the algorithm are

```
datatype 'label op = pop | swap 'label | push 'label 'label

type_synonym ('ctr_loc, 'label) rule =
  ('ctr_loc × 'label) × ('ctr_loc × 'label op)
type_synonym ('ctr_loc, 'label) conf = 'ctr_loc × 'label list

locale PDS = fixes Δ :: ('ctr_loc, 'label:: finite) rule set begin
  fun lbl where
    lbl pop = [] | lbl (swap γ) = [γ] | lbl (push γ γ') = [γ, γ']
  definition is_rule (infix ↪) where
    (p, γ) ↪ (p', w) ⟷ ((p, γ), (p', w)) ∈ Δ
  inductive_set step where
    (p, γ) ↪ (p', w) ⟶
      ((p, γ # w'), (), (p', lbl w @ w')) ∈ step
  interpretation LTS step .
end

datatype ('ctr_loc, 'noninit) state =
  Init 'ctr_loc | Noninit 'noninit

locale PDS_with_finals = PDS Δ
    for Δ :: ('ctr_loc :: enum, 'label :: finite) rule set +
    fixes F_inits :: 'ctr_loc set and F_noninits :: 'noninit set
begin
  definition finals = Init ` F_inits ∪ Noninit ` F_noninits
  definition inits = {q. ∃p. q = Init p}
  definition accepts A (p, w) =
    (∃q ∈ finals. (Init p, w, q) ∈ LTS.trans_star A)
  definition lang A = {c. accepts A c}
end
```

Fig. 2: The types and locales for pushdown systems

expressed as the premises of the implication shown in pre_star_rule's definition. The last two premises are taken directly from Schwoon's definition of the algorithm and the first one ensures that the transition we add into the new P-automaton is a new one. A single P-automaton can be related to different P-automata via pre_star_rule, which captures nondeterministic choice.

Consider the PDS defined by Δ in Fig. 3, and let the P-automaton A consist of the two solid transitions in the figure. Let A′ be A ∪ $\{(P_2, \gamma_2, P_0)\}$. Notice that $(P_2, \gamma_2, P_0) \notin$ A and $(p_2, \gamma_2) \hookrightarrow (p_0, \mathsf{pop})$ and $(P_0, \mathsf{lbl}\ \mathsf{pop}, P_0) \in$ LTS.trans_star A. From pre_star_rule's definition then follows that pre_star_rule A A′. Let A″ be A′ ∪ $\{(P_1, \gamma_1, Q_1\})$. From pre_star_rule's definition it follows that pre_star_rule A′ A″.

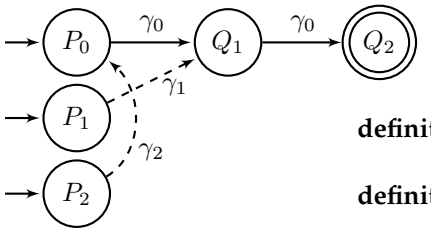


Fig. 3: Adding two transitions (dashed arrows) to a P-automaton. Initially (solid arrows) the P-automata encodes only configuration $(p_0, [\gamma_0, \gamma_0])$. After saturation, the configurations $(p_1, [\gamma_1, \gamma_0])$ and $(p_2, [\gamma_2, \gamma_0, \gamma_0])$ are also encoded.

We formalize what it means for a P-automaton A to be saturated w.r.t a rule *r*, and for A' to be a saturation of A:

definition saturated $r\ A = (\nexists A'.\ r\ A\ A')$

definition saturation $r\ A\ A' = (r^{**}\ A\ A' \wedge \text{saturated}\ r\ A')$

In our example, A'' is saturated and thus formally we have saturated pre_star_rule A'' and saturation pre_star_rule A A''.

We next prove the pre^* saturation algorithm correct. Here, we focus on the proof's most interesting aspects, especially those where we had to deviate from Schwoon's pen-and-paper proof, and refer to our formalization for full details [23].

The correctness theorem states that if a transition system A' is a saturation of a transition system A then the language of A' is indeed the pre^* closure of the language of A. Like Schwoon, we assume that the initial states are sources:

theorem *pre_star_rules_correct*:
assumes inits ⊆ LTS.srcs A
and saturation pre_star_rule A A'
shows lang A' = pre_star (lang A)

Schwoon's Lemma 3.1 is used to prove the $\supseteq$ direction of the theorem's conclusion. He proves it by considering an arbitrary predecessor configuration (p', w) of a configuration (p, v) in A's language. The proof proceeds by induction on the number of $\Rightarrow$ transitions from (p', w) to (p, v). We do not keep track of this number, but we instead prove the lemma by induction on the transitive and reflexive closure of $\Rightarrow$. The formalization of the proof is written in Isabelle's structured proof language Isar (not shown) and follows Schwoon's arguments. Schwoon's Lemma 3.2 is used to prove the $\subseteq$ direction of *pre_star_rules_correct*'s conclusion. We showcase Lemma 3.2 in Schwoon's formulation, but adapted to our notation:

Lemma 3.2 If saturation pre_star_rule A A' and $(p, w, q) \in$ LTS.trans_star A' then:

(a) $(p, w) \Rightarrow^* (p', w')$ for a configuration (p', w') such that $(p', w', q) \in A$;

(b) moreover, if q is an initial state, then $w' = []$.

In his proof, Schwoon claims to prove (a) by an induction and then that (b) will follow immediately from a simple argument. However, reading his proof we notice that he uses (b) in the proof of (a). We resolve this by noticing that we can strengthen (b) to hold for any stack w and not just the one w' claimed to exist in (a). Our formulation of (b) looks as follows:

lemma *word_into_init_empty*:
  assumes $(p, w, \mathsf{Init}\ q) \in \mathsf{LTS.trans\_star}\ A$
  and $\mathsf{inits} \subseteq \mathsf{LTS.srcs}\ A$
  shows $w = [] \wedge p = \mathsf{Init}\ q$

We prove (a) using the strengthened version of (b). Like Schwoon, we prove (a) by a nested induction. His outer induction is on the number of times the algorithm added transitions to the P-automaton. We instead prove the lemma by induction on the transitive reflexive closure of pre_star_rule. The inner induction is more challenging to formalize. Here, Schwoon considers a specific transition t which he defines as the ith transition added to P-automaton A. In the same context he considers a word w and two states, $\mathsf{Init}\ p$ and q, such that $(\mathsf{Init}\ p, w, q) \in \mathsf{LTS.trans\_star}\ A'$. He then defines j as the number of times t is used in $(\mathsf{Init}\ p, w, q) \in \mathsf{LTS.trans\_star}\ A'$. We may argue that this number is not well-defined, because there can be several paths from $\mathsf{Init}\ p$ to q consuming w, and on these paths t may not occur the same number of times. It turns out we can choose among these paths completely freely—any one of them will work, and so we just choose one arbitrarily. Formalizing this required us to define a variant of trans_star that keeps track of the intermediate states.

3.2 Nondeterministic $post^*$ Saturation

We call states with no incoming or outgoing transitions isolated. The $post^*$ saturation algorithm requires the addition of new noninitial states that are isolated in the automaton on which the algorithm is run. Under certain conditions the algorithm adds transitions into and out of these. Each such new state corresponds to a control location and a label. We extend the datatype of states with a new constructor Isolated for these:

datatype $('ctr\_loc, 'noninit, 'label)\ state =$
  $\mathsf{Init}\ 'ctr\_loc \mid \mathsf{Noninit}\ 'noninit \mid \mathsf{Isolated}\ 'ctr\_loc\ 'label$

Moreover, we define $\mathsf{isols} = \{q.\ \exists p.\ q = \mathsf{Isolated}\ p\}$.

Steps in the $post^*$ saturation are formalized as follows:

inductive post_star_rules **where**
$(p, \gamma) \hookrightarrow (p', \mathsf{pop}) \longrightarrow (\mathsf{Init}\ p', \varepsilon, q) \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
$\mathsf{post\_star\_rules}\ A\ (A \cup \{(\mathsf{Init}\ p', \varepsilon, q)\})$
| $(p, \gamma) \hookrightarrow (p', \mathsf{swap}\ \gamma') \longrightarrow (\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q) \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
$\mathsf{post\_star\_rules}\ A\ (A \cup \{(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q)\})$
| $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\ \gamma'') \longrightarrow$
$(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
$\mathsf{post\_star\_rules}\ A\ (A \cup (\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma'))$
| $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\gamma'') \longrightarrow$
$(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q) \notin A \longrightarrow$
$(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \in A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
$\mathsf{post\_star\_rules}\ A\ (A \cup \{(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)\})$

The relation has one rule for pop, one for swap, and two for push. It uses LTS_ε.trans_star_ε, which is similar to LTS.trans_star but allows ε-transitions that do not consume stack symbols. The transition $(\mathsf{Init}\ p', \varepsilon, q)$ is an ε-transition and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q)$ is a γ'-labeled non-ε-transition. The function lang_ε returns the language of a P-automaton with ε-transitions. We prove $post^*$ saturation correct:

theorem *post_star_rules_correct*:
assumes saturation post_star_rules $A\ A'$
and inits $\subseteq$ LTS.srcs A **and** isols $\subseteq$ LTS.isolated A
shows lang_ε $A' =$ post_star (lang_ε A)

Schwoon's definition of the $post^*$ rule has only one rule for push (in contrast to our two rules). In his rule, Schwoon *first* adds a transition $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$ and *then* adds a transition $(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)$. Consider his rule here presented in his formulation but our notation:

If $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\ \gamma'')$ and $(\mathsf{Init}\ p, \gamma, q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A$,
first add $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$;
then add $(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)$.

We were at first surprised that he specified this *first*/*then* order, but his correctness proof actually relies on it. Specifically, the order is used in his proof of Lemma 3.4, which is the key to prove the $\supseteq$ direction of *post_star_rules_correct*. We present Lemma 3.4 in Schwoon's formulation but our notation:

Lemma 3.4 If saturation post_star_rules $A\ A'$ and
$(\mathsf{Init}\ p, w, q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A'$ then:

(a) if $q \notin \mathsf{isols}$, then $(p', w') \Rightarrow^* (p, w)$ for a configuration (p', w') such that $(\mathsf{Init}\, p', w', q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A$;

(b) if $q = \mathsf{Isolated}\ p'\ \gamma'$, then $(p', \gamma') \Rightarrow^* (p, w)$.

Schwoon's proof is a nested induction. The outer induction is on the number of transitions $post^*$ has added. The induction step proceeds by an inner induction on the number of times the most recently added transition t was used in $(\mathsf{Init}\ p, w, q') \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A'$. (We resolve the ambiguity of that number's meaning in a similar way as for pre^*.) The proof then proceeds by a case distinction on which of the $post^*$ saturation rules added t. Consider the case where t was added by the "first" part of the rule for push. In this case, t has the form $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$. Schwoon states that "Then since $\mathsf{Isolated}\ p'\ \gamma'$ has no transitions leading into it initially, it cannot have played part in an application rule before this step, and t is the first transition leading to it. Also, there are no transitions leading away from t so far." Had Schwoon not forced the algorithm to *first* add the transition into $\mathsf{Isolated}\ p'\ \gamma'$ and *then* add the one out of it, then he could not have claimed that there are no transition leading away from t. We capture this idea in the following two lemmas, stating that if t is not present, then $\mathsf{Isolated}\ p'\ \gamma'$ must be a source and a sink:

lemma *post_star_rules_Isolated_source_invariant*:
assumes $\mathsf{post\_star\_rules}^{**}\ A\ A'$
and $\mathsf{isols} \subseteq \mathsf{LTS.isolated}\ A$
and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A'$
shows $\mathsf{Isolated}\ p'\ \gamma' \in \mathsf{LTS.srcs}\ A'$

lemma *post_star_rules_Isolated_sink_invariant*:
assumes $\mathsf{post\_star\_rules}^{**}\ A\ A'$
and $\mathsf{isols} \subseteq \mathsf{LTS.isolated}\ A$
and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A'$
shows $\mathsf{Isolated}\ p'\ \gamma' \in \mathsf{LTS.sinks}\ A'$

Formalizing Schwoon's push rule as a single rule in post_star_rules does not capture the order in which the two transition are added to the set. This is why we split the rule in two—one adding the transition into the new noninitial state and another adding the transition out of the new noninitial state. This does not yet impose the needed *first*/*then* order. However, we can impose the order by letting the latter rule be only applicable if the transition added by the former is indeed already in the automaton. This is possible because the transition added into state $\mathsf{Isolated}\ p'\ \gamma'$ is $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$, and thus we can refer to the states comprising this transition in any context where $\mathsf{Isolated}\ p'\ \gamma'$ is available, in particular, the second push rule. Note that our $post^*$ saturation algorithm is slightly more general than Schwoon's as we do not require the transition out of the new noninitial state to be added immediately after the transition into it, rather we allow this to happen at any time after.

fun (in LTS) reach **where**
  reach p [] = $\{p\}$
| reach p ($\gamma \# w$) = ($\bigcup q' \in (\bigcup (p', \gamma', q') \in$ step.
    if $p' = p \wedge \gamma' = \gamma$ then $\{q'\}$ else $\{\}$). reach q' w)

definition (in PDS) pre_star1 A = ($\bigcup((p, \gamma), (p', w)) \in \Delta$.
  $\bigcup q \in$ LTS.reach A (Init p') (lbl w). $\{$(Init p, γ, q)$\}$)

definition (in PDS) pre_star_exec = the $\circ$ while_option
  ($\lambda s.\ s \cup$ pre_star1 $s \neq s$) ($\lambda s.\ s \cup$ pre_star1 s)

Fig. 4: Executable pre^*

3.3 Combined $dual^*$ Saturation

We now consider the recent bi-directional search approach, called $dual^*$ [12]. With $dual^*$ we can check if the configurations of one P-automaton A_2 are reachable from another P-automaton A_1 by alternating between saturating A_2 towards its pre^* closure and A_1 towards its $post^*$ closure, while simultaneously (on-the-fly) keeping track of their intersection automaton. As soon as the intersection automaton becomes nonempty, we know that there is a state in A_2 that is reachable from A_1. This is the case even if the pre^* and $post^*$ automata are not saturated. Our correctness theorem is formalized here:

theorem *dual_star_correct_early_termination*:
  assumes inits $\subseteq$ LTS.srcs A_1 **and** inits $\subseteq$ LTS.srcs A_2
  and isols $\subseteq$ LTS.isolated $A_1 \cap$ LTS.isolated A_2
  and post_star_rules** A_1 A_1' **and** pre_star_rule** A_2 A_2'
  and lang_ε_inters (inters_ε A_1' (LTS_ε_of A_2')) $\neq \{\}$
  shows $\exists c_1 \in$ lang_ε A_1. $\exists c_2 \in$ lang A_2. $c_1 \Rightarrow^* c_2$

The function LTS_ε_of trivially converts a P-automaton to a P-automaton with ε-transitions. The function inters_ε calculates the intersection P-automaton with ε-transitions of two P-automata with ε-transitions using a product construction. The function lang_ε_inters gives the language of an intersection automaton. Since the $\subseteq$ directions of *pre_star_rule_correct* and *post_star_rules_correct* do not rely on A' being saturated we prove them assuming only respectively pre_star_rule** A_2 A_2' and post_star_rules** A_1 A_1' instead of saturation pre_star_rule A_2 A_2' and saturation post_star_rules A_1 A_1'. We use these more general lemmas to prove *dual_star_correct_early_termination*.

4 Executable Pushdown Reachability

To get an executable algorithm for pre^*, we resolve the nondeterminism by defining a functional program pre_star_exec, presented in Fig. 4 (where we

indicate the corresponding locale for each definition), with this characteristic property:

theorem *pre_star_exec_language_correct*:
assumes inits $\subseteq$ LTS.srcs A
shows lang (pre_star_exec A) = pre_star (lang A)

The function reach is trans_star's executable counterpart: for a state p and a word w, reach p w computes the set of states reachable from p via w using step (fixed in the LTS locale). In other words, we have $q \in$ reach p w iff $(p, w, q) \in$ trans_star.

The definition of pre_star_exec uses while_option, the functional while loop counterpart. Given a test predicate b, a loop body c and a loop state s, the expression while_option b c s computes the optional state Some $(c\ (\cdots (c\ (c\ s))))$ not satisfying b with the minimal number of applications of c, or None if no such state exists. Our specific loop keeps adding the results of a single step pre_star1 to the P-automaton comprising the loop state. We prove that our loop never returns None, i.e., it always terminates. We thus use the, defined partially as the (Some x) = x, in pre_star_exec to extract the resulting P-automaton. The step pre_star1 computes the set of all transitions that can be added by a single application of pre_star_rule.

Fig. 4's definitions are executable: Isabelle can interpret them as functional programs and extract Standard ML, Haskell, OCaml, or Scala code [46], but it is usually not possible to extract code for inductive predicates (such as trans_star or the transitive closure in saturation) or definitions involving quantifiers ranging over an infinite domain (as in saturated). The definition of pre_star_exec has an obvious inefficiency. In every iteration, pre_star1 is evaluated twice: once as a part of the loop body and once as a part of the test. Instead we use the following improved equation, which replaces while_option with explicit recursion, for code extraction.

lemma *pre_star_exec_code*[code]:
pre_star_exec s = (let s' = pre_star1 s in
if $s' \subseteq s$ then s else pre_star_exec $(s \cup s')$)

With the executable algorithm for pre^*, we decide the reachability problem for P-automata using the check function shown in Fig. 5. It inputs a PDS Δ along with two P-automata represented by their transition relations (A_1 and A_2), their final initial states (F_1 and F_2) and their final noninitial states (F_1^{ni} and F_2^{ni}). The computation proceeds by intersecting (inters) the initial P-automaton with the pre^* saturation of the final P-automaton and checking the result's nonemptiness (nonempty). Fig. 5 refers to functions pre_star_exec, inits, finals, and trans_star which we introduced earlier in the context of different locales, outside of the respective locale. Therefore, these functions take additional parameters that correspond to the fixed parameters of the

definition nonempty $A\ P\ Q =$
$(\exists p \in P.\ \exists q \in Q.\ \exists w.\ (p, w, q) \in \mathsf{trans\_star}\ A)$

definition inters $A\ B =$
$\{((p_1, p_2), w, (q_1, q_2)).\ (p_1, w, q_1) \in A \wedge (p_2, w, q_2) \in B\}$

definition nonempty_inter $\Delta\ A_1\ F_1\ F_1^{ni}\ A_2\ F_2\ F_2^{ni} =$
nonempty (inters A_1 (pre_star_exec $\Delta\ A_2$))
$((\lambda x.\ (x, x))$ ‘ inits) (finals $F_1\ F_1^{ni} \times$ finals $F_2\ F_2^{ni}$)

definition check $\Delta\ A_1\ F_1\ F_1^{ni}\ A_2\ F_2\ F_2^{ni} =$
(if $\neg$inits $\subseteq$ LTS.srcs A_2 then None
else Some (nonempty_inter $\Delta\ A_1\ F_1\ F_1^{ni}\ A_2\ F_2\ F_2^{ni}$)

Fig. 5: Reachability check for P-automata

respective locale if they are used by the function (e.g., we write pre_star_exec Δ instead of pre_star_exec for an implicitly fixed Δ).

The definition of nonempty is not executable because of the quantification over words w. We implement, but omit here, the straightforward executable algorithm that starts with the set of initial states P and iteratively adds transitions from A until it reaches Q or saturates without reaching Q, in which case the language is empty since no state in Q is reachable from P.

Overall, check returns an optional Boolean value, where None signifies a well-formedness violation on the final P-automaton: a non-source initial state in A_2. If check returns Some b, then b is the answer to the reachability problem for P-automata. We formalize this characterization of check by the following two theorems (phrased outside of locales).

theorem *check_None*:
check $\Delta\ A_1\ F_1\ F_1^{ni}\ A_2\ F_2\ F_2^{ni} =$ None $\longleftrightarrow$
$\neg$inits $\subseteq$ LTS.srcs A_2

theorem *check_Some*:
check $\Delta\ A_1\ F_1\ F_1^{ni}\ A_2\ F_2\ F_2^{ni} =$ Some $b \longleftrightarrow$
(inits $\subseteq$ LTS.srcs $A_2 \wedge (b \longleftrightarrow$
$(\exists p\ w\ p'\ w'.$ step_starp $\Delta\ (p, w)\ (p', w') \wedge$
$(p, w) \in$ lang $A_1\ F_1\ F_1^{ni} \wedge (p', w') \in$ lang $A_2\ F_2\ F_2^{ni})))$

5 Differential Testing

Differential testing [19–21] is a technique for finding implementation errors by executing different algorithms solving the same problem on a set of test cases and comparing the outputs. Differential testing has been effective for finding errors in a wide range of domains, from network certificate validation [47] to

JVM implementations [48]. Yet, even different algorithms do not necessarily fail independently, e.g., when built from the same specification [49] or when sharing potentially faulty components, e.g., input parsers or preprocessing. To reduce the danger of missing such errors, we suggest to incorporate a formally verified implementation in differential testing. Moreover, in case of a discrepancy the verified oracle reliably tells us which of the unverified implementations is wrong.

5.1 Differential Testing of Pushdown Reachability

Our executable formalization of pushdown reachability allows us to perform differential testing on unverified tools for the same problem. A test case for pushdown reachability consists of a PDS with rules Δ and two P-automata $\mathcal{A}_1$ and $\mathcal{A}_2$ representing the initial and final configurations of interest. The answer to the test case is whether there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ.

To execute the formalization on a given test case, we generate an Isabelle theory file, which first defines the control locations, labels, and automata states as finite subsets of the natural numbers (their sizes depending on the specific test case), and then includes for the pushdown rules Δ and the two P-automata, each represented by its transitions A_i along with the accepting (initial and noninitial) states F_i and $\mathsf{F}_i^{\mathsf{ni}}$ for $i \in \{1, 2\}$. Fig. 3 shows a specific example of Δ and A definitions.

We generate a lemma that uses our check function, where the expected result Some True or Some False is inserted depending on the answer produced by an unverified tool under test (invoked before generating the theory on the same inputs):

lemma check Δ A_1 F_1 $\mathsf{F}_1^{\mathsf{ni}}$ A_2 F_2 $\mathsf{F}_2^{\mathsf{ni}}$ = Some True **by** eval

The eval proof method extracts Standard ML code for check and other constants in the lemma and executes the lemma statement as an expression. It succeeds iff the lemma evaluates to True. We call a test case a counter-example, if the proof method fails. One could also run the extracted code outside Isabelle, but our setup allows us to generate the inputs to check on the formalization level instead of that of the extracted code.

To efficiently check a large number of test cases, we batch multiple definitions and lemmas into one theory file, thus reducing the overhead of starting Isabelle. We run Isabelle from the command line and check the output log for any failing eval proofs, which correspond to failing test cases.

Algorithm 1 Specialization of delta-debugging [22] to PDS.

Input: Reachability tools $tool$ and $oracle$, PDS (P, Γ, Δ),
P-automata $\mathcal{A}_i = (P \cup N_i, \rightarrow_i, P, F_i)$ for $i \in \{1, 2\}$.
Output: Minimal counter-example (failing testcase)

1: $c \leftarrow \Delta \cup (\{1\} \times (\rightarrow_1 \cup F_1)) \cup (\{2\} \times (\rightarrow_2 \cup F_2))$ ▷ *Convert to a set of features*
2: **return** DD(c, 2) ▷ *returned set of features can be converted to PDS and P-automata as on lines 10-11*

3: **function** DD(c, n) ▷ *c is a test case, n is granularity*
4: let $c_1 \uplus \cdots \uplus c_n = c$, all c_i as evenly sized as possible
5: **if** $\exists i$. BAD(c_i) **return** DD(c_i, 2)
6: **else if** $\exists i$. BAD($c \setminus c_i$) **return** DD($c \setminus c_i$, $\max(n-1, 2)$)
7: **else if** $n < |c|$ **return** DD(c, $\min(2n, |c|)$)
8: **else return** c

9: **function** BAD(c) ▷ *c is a test case*
10: let $\Delta' = c \cap \Delta$ ▷ *extract PDS rules and P-automata*
11: for $i \in \{1, 2\}$ let $\mathcal{A}'_i = (P \cup N_i, \rightarrow'_i, P, F'_i)$ where $\rightarrow'_i = \{t \in \rightarrow_i \mid (i, t) \in c\}$ and $F'_i = \{q \in F_i \mid (i, q) \in c\}$
12: with both tools check if $\mathcal{A}'_1$ reaches $\mathcal{A}'_2$ via (P, Γ, Δ')
13: **return** *false* if $tool$ and $oracle$ agree, else *true*

5.2 Automatic Counter-Example Minimization

If differential testing finds a failing test case, we use delta-debugging [22] to automatically reduce it to a minimal failing test case to help the subsequent debugging process. We use the minimizing delta debugging algorithm [22] that sees a test case as a set of features, and works by systematically testing different subsets until a minimal failing test case is found.

We use delta debugging on any discovered counter-example and fix the set of features to contain: (i) each pushdown rule, (ii) each transition in either of the P-automata, and (iii) each final state in a P-automaton (as opposed to it not being final).

States and labels are identified by unique names, and the initial P-automata states are exactly the states mentioned in any pushdown rule in the feature set. We specialize the general delta debugging algorithm to pushdown systems as shown in Algorithm 1. The algorithm first creates the set of features and calls the recursive function DD with this set of features and the granularity 2. The function then splits the set of features into a number of equally sized subsets (according to the granularity) and checks if any of these subsets or their complements still fail. If yes, then the function tries to recursively reduce the set of features further, otherwise it will increase the granularity

$\Delta = \{$
$(p_0, C) \hookrightarrow (p_3, \text{swap } E)$, $(p_2, B) \hookrightarrow (p_0, \text{push } A\ B)$, $(p_2, B) \hookrightarrow (p_3, \text{push } C\ B)$,
$(p_0, D) \hookrightarrow (p_0, \text{swap } A)$, $(p_1, B) \hookrightarrow (p_0, \text{swap } C)$, $(p_2, A) \hookrightarrow (p_0, \text{push } C\ A)$, $(p_3, D) \hookrightarrow (p_0, \text{push } B\ D)$,
$(p_0, E) \hookrightarrow (p_0, \text{push } B\ E)$, $(p_1, D) \hookrightarrow (p_0, \text{swap } C)$, $(p_2, C) \hookrightarrow (p_0, \text{push } C\ C)$, $(p_3, C) \hookrightarrow (p_0, \text{push } E\ C)$,
$(p_0, D) \hookrightarrow (p_0, \text{push } D\ D)$, $(p_1, C) \hookrightarrow (p_0, \text{swap } B)$, $(p_2, D) \hookrightarrow (p_0, \text{push } B\ D)$, $(p_3, C) \hookrightarrow (p_0, \text{swap } E)$,
$(p_0, D) \hookrightarrow (p_0, \text{pop})$, $(p_1, C) \hookrightarrow (p_0, \text{swap } E)$, $(p_2, C) \hookrightarrow (p_1, \text{push } C\ C)$, $(p_3, C) \hookrightarrow (p_1, \text{push } A\ C)$,
$(p_0, D) \hookrightarrow (p_1, \text{swap } A)$, $(p_1, B) \hookrightarrow (p_1, \text{swap } C)$, $(p_2, A) \hookrightarrow (p_1, \text{push } B\ A)$, $(p_3, B) \hookrightarrow (p_1, \text{pop})$,
$(p_0, A) \hookrightarrow (p_1, \text{push } C\ A)$, $(p_1, E) \hookrightarrow (p_1, \text{swap } C)$, $(p_2, A) \hookrightarrow (p_2, \text{push } A\ A)$, $(p_3, E) \hookrightarrow (p_2, \text{swap } C)$,
$(p_0, E) \hookrightarrow (p_2, \text{push } A\ E)$, $(p_1, A) \hookrightarrow (p_2, \text{swap } A)$, $(p_2, C) \hookrightarrow (p_2, \text{swap } A)$, $(p_3, B) \hookrightarrow (p_2, \text{push } D\ B)$,
$(p_0, B) \hookrightarrow (p_2, \text{push } D\ B)$, $(p_1, D) \hookrightarrow (p_2, \text{swap } D)$, $(p_2, E) \hookrightarrow (p_2, \text{swap } A)$, $(p_3, E) \hookrightarrow (p_3, \text{swap } A)$,
$(p_0, C) \hookrightarrow (p_2, \text{swap } D)$, $(p_1, C) \hookrightarrow (p_2, \text{swap } E)$, $(p_2, A) \hookrightarrow (p_2, \text{push } B\ A)$, $(p_3, A) \hookrightarrow (p_3, \text{push } C\ A)$,
$(p_0, E) \hookrightarrow (p_2, \text{swap } E)$, $(p_1, C) \hookrightarrow (p_3, \text{swap } D)$, $(p_2, B) \hookrightarrow (p_2, \text{swap } E)$, $(p_3, E) \hookrightarrow (p_3, \text{swap } D)$,
$(p_0, E) \hookrightarrow (p_3, \text{push } B\ E)$, $(p_1, D) \hookrightarrow (p_3, \text{pop})$, $(p_2, E) \hookrightarrow (p_3, \text{push } A\ E)$, $(p_3, C) \hookrightarrow (p_3, \text{pop})\}$

$A_1 = \{(\text{Init } p_0, B, \text{Noninit } q_1), (\text{Init } p_0, D, \text{Noninit } q_0),$
$(\text{Init } p_2, B, \text{Noninit } q_0), (\text{Init } p_3, A, \text{Noninit } q_2),$
$(\text{Noninit } q_0, D, \text{Noninit } q_1), (\text{Noninit } q_2, C, \text{Noninit } q_0)\}$
$F_1 = \{\}$ $F_1^{ni} = \{q_1\}$
$A_2 = \{(\text{Init } p_2,\ A,\ \text{Noninit } q_0)), (\text{Init } p_2,\ B,\ \text{Noninit } q_0)\}$
$F_2 = \{p_0, p_2\}$ $F_2^{ni} = \{\}$

$\Delta = \{(p_0, D) \hookrightarrow (p_0, \text{pop})\}$
$A_1 = \{(\text{Init } p_0, D, \text{Noninit } q_0),$
$(\text{Noninit } q_0, D, \text{Noninit } q_1)\}$
$F_1 = \{\}$ $F_1^{ni} = \{q_1\}$
$A_2 = \{\}$
$F_2 = \{p_0\}$ $F_2^{ni} = \{\}$

Fig. 6: Original and minimized (bottom right) counter-example

and try again. The function BAD converts the set of features into a reduced pushdown system and two reduced P-automata and checks if the given tool implementation is still inconsistent with the oracle. We note that minimal failing counter-examples are only locally minimal and not necessarily unique. Yet, minimization is effective and necessary. Fig. 6 shows a real bug example we discovered by random differential testing in the PDAAAL library for pushdown reachability [12] and its minimization by Algorithm 1.

6 Case Study: Analysis of PDAAAL

We apply differential testing with automatic counter-example minimization to PDAAAL [50], a recent C++ implementation of pushdown reachability checking, which appears to be the currently most efficient library for pushdown reachability [12]. PDAAAL implements $post^*$, pre^* and $dual^*$ [12].

These three different algorithms can be used in classical differential testing without a verified oracle, but given the large amount of shared code this is bound to miss some errors. And without a verified oracle, manual effort is needed to determine which implementation is faulty in case of discrepancies. This motivates using our verified reachability check via pre^*, and we compare the output of each unverified algorithm to the output of our trustworthy oracle on a large number of test cases.

6.1 Methodology of Test Case Generation

We structure our test case generation in three phases.

In phase one, we use real-world tests generated from the domain of network verification, which PDAAAL was originally built for as a backend [9]. We generate pushdown reachability problems from realistic network verification use-cases on (up to) 100 random reachability queries on each of the 260 different networks derived from the Internet Topology Zoo [51] giving a total of $25\,512$ test cases.

In phase two, we randomly generate valid pushdown systems and P-automata. We generate $15\,000$ cases of varying sizes with 4 control locations, 5 labels, up to 200 pushdown rules, and up to 13 automata transitions. Our generator writes all ingredients (pushdown system and P-automata) to a JSON file, which is then translated to the Isabelle definitions and correctness lemmas that incorporate the unverified answers.

Finally, in phase three, we exhaustively enumerate the set of all test cases up to a certain (small) size. For the pushdown systems $|P| = |\Gamma| = 2$ and $|\Delta| \leq 2$, and for P-automata $|N_1| = 2$, $|N_2| = 1$ and $|{\rightarrow}| \leq 2$. We remove symmetric cases, where swapping state names or labels gives an identical case. In total, this yields close to 27 million combinations of pushdown systems and P-automata. For the exhaustive tests, we output both JSON files and Isabelle definitions directly from the test case generator. A bash script stitches together the Isabelle definitions into a single theory file with a batch of test cases to benefit from Isabelle's parallel processing of proofs.

6.2 Results

The real-world test cases showed no discrepancies between the verified oracle and PDAAAL. This indicates that PDAAAL has already been thoroughly tested on this type of problem instances. Isabelle ran out of memory in 30 of the $25\,512$ test cases. The average CPU time (on AMD EPYC 7642 processors at 1.5 GHz) per test case was 35 seconds for Isabelle, while PDAAAL used less than 0.02 seconds on most cases.

Phase two, however, resulted in $1\,334$ discrepancies. By applying our counter-example minimization, we noted that all these cases had a common trait: the P-automaton $\mathcal{A}_2$ accepted the empty word. This helped us find the first implementation error in the implementation of the on-the-fly automata intersection when using $post^*$. The $post^*$ algorithm can introduce ε-transitions, which were not handled correctly by the intersection implementation. In most cases, this does not matter, as for any ε-transition followed by a normal transition the $post^*$ algorithm adds a direct transition at some later point. However, in the case of an empty stack being accepted by $\mathcal{A}_2$, this does not happen, which causes the unverified algorithm to return the wrong answer

10: **function** AddTransition($q_i \xrightarrow{\gamma}_i q_i'$) ▷ *with* $i \in \{1, 2\}$
11: add $q_i \xrightarrow{\gamma}_i q_i'$ to $\mathcal{A}_i$
12: **for all** $q_{3-i}, q_{3-i}' \in Q_{3-i}$ s.t. $(q_1, q_2) \in R$ and $q_{3-i} \xrightarrow{\gamma}_{3-i} q_{3-i}'$ **do**
13: add $(q_1, q_2) \xrightarrow{\gamma} (q_1', q_2')$ to $\mathcal{A}_\cap$
14: AddState(q_1', q_2')

(a) Snippet of (correct) intersection pseudocode by Jensen et al. [12]

src/include/pdaaal/Solver.h

```
@@ -119,10 +119,10 @@ namespace pdaaal {
119 119                 if (res.second) { // New edge is not already in edges (rel U workset).
120 120                     _workset.emplace(from, label, to);
121 121                     if (trace != nullptr) { // Don't add existing edges
    122 +                       _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
122 123                         if constexpr (ET) {
123 124                             _found = _found || _early_termination(from, label, to, trace_ptr_from<W>(trace));
124 125                         }
125     -                       _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
126 126                     }
127 127                 }
128 128             };
```

(b) PDAAAL's C++ code showing the resolution of the second error

Fig. 7: Discovered second implementation error and its correct pseudocode

False. We resolved the error and re-ran the generated tests. After that only one discrepancy remained.

This second error was found in the implementation of pre^*. The minimized counter-example helped us find the source of the implementation error: the set of automata transitions was updated only after calling the function that performs the nonemptiness check of the intersection automaton, but it should have been updated before that call. We argue that this error is subtle, as it only causes a single failure out of 15 000 randomly generated test cases. Fig. 7a shows the correct pseudocode by Jensen et al. [12]. Fig. 7b shows PDAAAL's corresponding C++ code and the change resolving the error, where the line that needed to be moved corresponds to the pseudocode's Line 11.

For both errors, the affected test cases resulted in a correct answer for at least one of the other search strategies in PDAAAL. This is not the case for the last error, which is found in code shared by all three methods, and where PDAAAL's algorithms disagree only with Isabelle. This error is caused by a mismatch between the assumptions of the parser that builds the pushdown system and the data structure that stores the pushdown rules. The parser assumes that it can incrementally add rules to the data structure without knowing all labels in advance, but the data structure assumes to know all labels from the start to implement a memory optimization that replaces a rule that applies to all labels by a wildcard.

For the first two test phases, the program that generated Isabelle definitions also depended on this parser, so the bug was not discovered until the third phase, which has a different setup. After the three bugs were fixed, all test cases pass.

7 Conclusion

We presented a methodology that increases the reliability of tools and libraries for pushdown reachability analysis. To this end, we formalized and proved in Isabelle/HOL the correctness of the essential saturation algorithms used in such tools. We extracted an executable program from our formalization and used it as a trustworthy oracle for differential testing. Putting the modern pushdown analysis library PDAAAL on the testbed, we discovered a number of implementation errors in its code, even though the library performed flawlessly in its application domain. Using our automatic counter-example minimization based on delta-debugging, we were able to identify the sources of these errors and suggested fixes to PDAAAL's implementation that now passes all the differential tests.

This process significantly increased PDAAAL's reliability and shows that with a moderate effort, the combination of proof assistants with code generation, differential testing, and delta-debugging is highly fruitful. The execution of all tests in the three phases took 303 CPU days. We executed the tests on a compute cluster with 1 536 CPU cores. The formalization work took about two person-months for experienced formalizers, creating about 4 400 nonempty lines of Isabelle definition and proofs. An additional half person-month of work was needed to implement the differential testing and counter-example minimization, set up the tests, and localize and resolve the discovered errors. This one-time effort will also benefit the future development of PDAAAL.

Too often, the race for better performance can lead to subtle implementation errors. Our methodology shows how formally verified algorithms that were not tuned for performance can be used to improve the quality of tuned but unverified algorithms.

Acknowledgements. This research was supported by the Independent Research Fund Denmark (DFF project QASNET) and by Novo Nordisk Fonden (NNF20OC0063462).

References

[1] J. R. Büchi, "Regular canonical systems," *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[2] J. Esparza and J. Knoop, "An automata-theoretic approach to interprocedural data-flow analysis," in *FOSSACS'99*, ser. LNCS, vol. 1578. Springer, 1999, pp. 14–30.

[3] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *CAV'05*, ser. LNCS, vol. 3576. Springer, 2005, pp. 449–461.

[4] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 324–336.

[5] S. Schwoon, "Moped," in *http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/*, 2002.

[6] D. Suwimonteerabuth, S. Schwoon, and J. Esparza, "jMoped: A java bytecode checker based on Moped," in *TACAS'05*, ser. LNCS, vol. 3440. Springer, 2005, pp. 541–545.

[7] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[8] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, pp. 217–227.

[9] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*, 2020, pp. 474–481.

[10] I. v. Duijn, P. G. Jensen, J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "Automata-theoretic approach to verification of MPLS networks under link failures," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 766–781, 2022.

[11] N. Kidd, A. Lal, and T. Reps, "WALi: The weighted automaton library," 2007. [Online]. Available: https://research.cs.wisc.edu/wpis/wpds/wali/

[12] P. G. Jensen, S. Schmid, M. K. Schou, J. Srba, J. Vanerio, and I. van Duijn, "Faster pushdown reachability analysis with applications in network verification," in *Automated Technology for Verification and Analysis (ATVA 2021)*, ser. LNCS, vol. 12971. Springer, 2021, pp. 170–186.

[13] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps, "Compositional recurrence analysis revisited," in *Proc. ACM PLDI*, 2017, pp. 248–262.

[14] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An inter-procedural static analysis framework for C/C++," in *TACAS'19*, ser. LNCS, vol. 11428. Springer, 2019, pp. 393–410.

[15] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 69–87.

[16] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proc. ACM POPL*, 2016, pp. 499–512.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[18] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, 2002.

[19] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[20] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *ESEC-FSE 2007*. ACM, 2007, pp. 549–552.

[21] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *ICSE 2007*. IEEE Computer Society, 2007, pp. 621–631.

[22] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[23] A. Schlichtkrull, M. K. Schou, J. Srba, and D. Traytel, "Repeatability package for "Differential testing of pushdown reachability with a formally verified oracle"," Zenodo, 2022.

[24] J. Schneider, D. A. Basin, S. Krstic, and D. Traytel, "A formally verified monitor for metric first-order temporal logic," in *RV 2019*, ser. LNCS, vol. 11757. Springer, 2019, pp. 310–328.

[25] D. A. Basin, T. Dardinier, L. Heimes, S. Krstic, M. Raszyk, J. Schneider, and D. Traytel, "A formally verified, optimized monitor for metric first-order dynamic logic," in *IJCAR 2020*, ser. LNCS, N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166. Springer, 2020, pp. 432–453.

[26] A. Schlichtkrull, J. C. Blanchette, and D. Traytel, "A verified prover based on ordered resolution," in *CPP 2019*, A. Mahboubi and M. O. Myreen, Eds. ACM, 2019, pp. 152–165.

[27] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[28] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, "Ironfleet: proving safety and liveness of practical distributed systems," *Commun. ACM*, vol. 60, no. 7, pp. 83–92, 2017.

[29] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proc. ACM PLDI*, 2015, pp. 357–368.

[30] M. Lesani, C. J. Bell, and A. Chlipala, "Chapar: certified causally consistent distributed key-value stores," in *Proc. ACM POPL*, 2016, pp. 357–370.

[31] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. ACM PLDI*, 2011, pp. 283–294.

[32] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *EuroSys 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds. ACM, 2017, pp. 328–343.

[33] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus, "A fully verified executable LTL model checker," in *CAV 2013*, ser. LNCS, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 463–478.

[34] S. Wimmer, "Munta: A verified model checker for timed automata," in *FORMATS 2019*, ser. LNCS, É. André and M. Stoelinga, Eds., vol. 11750. Springer, 2019, pp. 236–243.

[35] T. Braibant and D. Pous, "Deciding Kleene algebras in Coq," *Log. Methods Comput. Sci.*, vol. 8, no. 1, 2012.

[36] P. Lammich and T. Tuerk, "Applying data refinement for monadic programs to Hopcroft's algorithm," in *ITP 2012*, ser. LNCS, L. Beringer and A. P. Felty, Eds., vol. 7406. Springer, 2012, pp. 166–182.

[37] D. Jiang and W. Li, "The verification of conversion algorithms between finite automata," *Sci. China Inf. Sci.*, vol. 61, no. 2, pp. 028 101:1–028 101:3, 2018.

[38] S. Berghofer and M. Reiter, "Formalizing the logic-automaton connection," in *TPHOLs 2009*, ser. LNCS, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674. Springer, 2009, pp. 147–163.

[39] Y. Minamide, "Verified decision procedures on context-free grammars," in *TPHOLs 2007*, ser. LNCS, K. Schneider and J. Brandt, Eds., vol. 4732. Springer, 2007, pp. 173–188.

[40] M. V. M. Ramos, J. C. B. Almeida, N. Moreira, and R. J. G. B. de Queiroz, "Formalization of the pumping lemma for context-free languages," *J. Formaliz. Reason.*, vol. 9, no. 2, pp. 53–68, 2016.

[41] P. Lammich, M. Müller-Olm, and A. Wenner, "Predecessor sets of dynamic pushdown networks with tree-regular constraints," in *CAV 2009*, ser. LNCS, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 525–539.

[42] P. Lammich, "Formalization of dynamic pushdown networks in Isabelle/HOL," 2009. [Online]. Available: https://www21.in.tum.de/~lammich/isabelle/dpn-document.pdf

[43] A. Bouajjani, M. Müller-Olm, and T. Touili, "Regular symbolic analysis of dynamic networks of pushdown systems," in *CONCUR 2005*, ser. LNCS, M. Abadi and L. de Alfaro, Eds., vol. 3653. Springer, 2005, pp. 473–487.

[44] T. Nipkow and G. Klein, *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[45] C. Ballarin, "Locales: A module system for mathematical theories," *J. Autom. Reason.*, vol. 52, no. 2, pp. 123–153, 2014.

[46] F. Haftmann and T. Nipkow, "Code generation via higher-order rewrite systems," in *FLOPS 2010*, ser. LNCS, M. Blume, N. Kobayashi, and G. Vidal, Eds., vol. 6009. Springer, 2010, pp. 103–117.

[47] C. Tian, C. Chen, Z. Duan, and L. Zhao, "Differential testing of certificate validation in SSL/TLS implementations: An RFC-guided approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 24:1–24:37, 2019.

[48] Y. Chen, T. Su, and Z. Su, "Deep differential testing of JVM implementations," in *ICSE 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 1257–1268.

[49] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Eng.*, vol. 12, no. 1, pp. 96–109, 1986.

[50] M. K. Schou, P. G. Jensen, D. Kristiansen, and B. C. Schrenk, "PDAAAL," GitHub, 2021. [Online]. Available: https://github.com/DEIS-Tools/PDAAAL

[51] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

Paper G

Discovery of Flow Splitting Ratios in ISP Networks with Measurement Noise

Morten Konggaard Schou, Ingmar Poese, and Jiří Srba

The paper is under submission.

The layout has been revised.

Abstract

Network telemetry and analytics is essential for providing highly dependable services in modern computer networks. In particular, network flow analytics for ISP networks allows operators to inspect and reason about traffic patterns in their networks in order to react to anomalies. High performance network analytics systems are designed with scalability in mind, and can consequently only observe partial information about the network traffic. Still, they need to provide a holistic view of the traffic, including the distribution of different traffic flows on each link. It is impractical to monitor such fine-grained telemetry, and in large, heterogeneous networks it is often too complex and error-prone, if not impossible, to access and maintain all technical specifications and router-specific configurations needed to determine e.g. the load balancing weights used when traffic is split onto multiple paths. The ratios by which flows are split on the possible paths must be derived indirectly from the measured flow demands and link utilizations. Motivated by a case study provided by a major European ISP, we suggest an efficient method to estimate the flow splitting ratios. Our approach, based on quadratic linear programming, is scalable and robust to the measurement noise found in a typical network analytics deployment. Finally, we implement an automated tool for estimating the flow splitting ratios and document its applicability on real data from the ISP.

1 Introduction

Network flow analytics [1, 2] in internet service provider (ISP) networks is often employed by network operators for monitoring the traffic patterns [3–5]. This can help to optimize overall network performance and link utilizations. As modern computer networks transfer huge quantities of data, it is impossible to store and analyze every single packet forwarded in the network. By packet sampling (using e.g. NetFlow [6] or IPFIX [7]), a network operator is though capable of estimating, with a relatively high precision, the number of packets transferred by each flow in the network. Similarly, packet counters for each interface can provide reliable information on the current link utilizations. However, answering questions like: "*What traffic caused a spike on this link yesterday?*" requires the analytics to not only show the total traffic dispatched on each link (identifying the spike), but it also needs to break down this traffic into the different flows, in order to determine where the anomaly originates from.

In this paper, we tackle the problem of correlating flows with link traffic in a practical and scalable manner—a problem arising from a case study with a major network analytics company that monitors over 6000 routers across multiple ISPs.

Sampling packet headers on every link in the network can answer such

questions; however, it has severe scalability issues. Instead, current high performing network analytics systems sample packet headers at the ingress routers only and combine this with the information from Border Gateway Protocol (BGP) [8] and the interior gateway protocol (IGP) (e.g. IS-IS [9, 10] or OSPF [11]) to determine the links traversed by the packet.

A lookup of the packet's destination in the ingress routing table determines the BGP next-hop, which is the router where the packet will egress the network. The possible paths that the packet can use to go through the network from ingress to egress are obtained from the IGP. All the packets that travel from the same ingress to egress in the network are aggregated into a *flow* that has a certain demand (size in bytes per second) averaged over some time window. Figure 1a shows an example network where links are annotated by the current link utilizations and Figure 1b depicts two flows of demand 12 and 4, respectively. In order to better distribute traffic along the links and thus reduce the maximum link utilization [12], flows can be split along multiple paths as demonstrated in the last column in Figure 1b. The splitting ratios can be uniform among the available paths, or they can depend on the link capacities [13] or custom link weights as shown in Figure 1c.

In practice the flow splitting ratios on the router depend on many technical, vendor-specific implementation details and configurations—some of which may not be accessible. Obtaining and processing this fine-grained information across a large heterogeneous network would require a very complex system. Hence the network analytics company deem it impossible in practice to obtain the flow splitting ratios directly from the router. As it is, moreover, infeasible to sample and categorize the packets traversing each link to the corresponding flows, we need additional information in order to first infer the flow splitting ratios and then estimate how much of each flow contributes to the load on a concrete link in the network.

Fortunately, each router has a byte counter for each interface that measures the total amount of traffic sent out on each link. This information is regularly queried using SNMP [14, 15], and then the link utilization for a given time interval is estimated by linear interpolation between SNMP measurements. In our example from Figure 1a, each link is annotated with the current link utilization. We now want to solve the following *Flow Splitting Ratios* (FSR) problem: given flow demands, their paths and aggregated link utilizations, find the flow splitting ratios such that when we accordingly project flow demands onto the links, the predicted traffic on each link matches (as close as possible) the measured link utilizations.

As our main contribution, we provide a practical and efficient solution to the FSR problem, employing quadratic linear programming. As a concrete instance of the FSR problem, consider our running example from Figure 1: given the flow demands, available paths and the link utilizations, our approach automatically predicts the splitting ratios at each node (depicted in the last

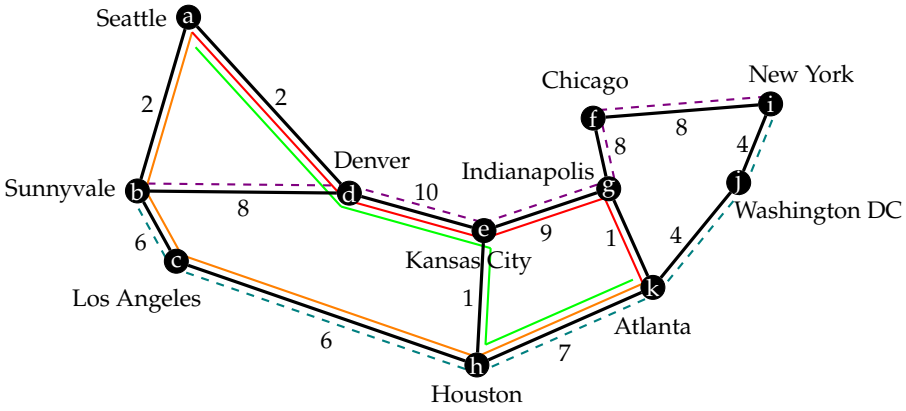
column of Figure 1c) and hence identifies how much every flow contributes to the total load on each link. Moreover, we suggest a filtering method to compute the splitting ratios even in case of large (but relatively rare) measurement errors that are present in a practical deployment of a real network. The suggested mechanism can efficiently deal with such measurement noise and errors and we demonstrate the robustness of our approach on a large set of simulated networks from the Topology Zoo [16] as well as on real traffic data from a major European ISP (in collaboration with a network analytics company)

We observe that our approach achieves high precision in determining the load balancing weights even in cases where the measured data are imprecise and occasionally significantly deviate from the actual ingress traffic. Based on an extensive statistical evaluation on a benchmark of over 190 real-world ISP topologies, we conclude that our filtering technique helps to improve the precision by an order of magnitude in the best cases and achieves about 66% improvement in the median case. Our approach scales to even large ISP networks with thousands of routers and millions of flows. This allows us to analyze real traffic data from a major European ISP network (which consists of over 3.000 routers and 14.000 links) in a matter of minutes. We automatically identify the load balancing weights in this network (which in this concrete case closely correlates with the capacity-based splitting ratios where the balancing weights are proportional to the link capacities) and put the more precise flow analysis into production (as a part of a network analysis tool developed by the company).

Related Work There are several network approaches based on linear programming (see e.g. [17–20]) that compute/synthesize optimal splitting ratios for traffic engineering and congestion-free resilience. We, on the other hand, use LP to reverse engineer splitting ratios employed in a real network with the purpose of providing more accurate traffic flow analytics. Contrary to other papers that use linear objective functions, we employ quadratic optimization that is better suited for this application domain. From the network monitoring research, network traffic analysis and visualization tools like NVisionIP [21], Flowyager [22] and VITALflow [23] have been designed for the purpose of network security [21, 24, 25] and management [22, 23]. To the best of our knowledge, none of these tools can reliably project the flow traffic on each link, unless making assumptions on the underlying router configurations, which can be difficult to obtain for larger networks.

2 Problem Formalization

In this section, we shall first define the notion of a network and traffic flows and then formally rephrase the problem of identifying the flow splitting ratios.



(a) Abilene network from the Internet Topology Zoo [16]

Flow	Demand	Paths
Sunnyvale → New York	12	b-d-e-g-f-i b-c-h-k-j-i
Seattle → Atlanta	4	a-d-e-g-k a-d-e-h-k a-b-c-h-k

(b) Two flows in Abilene network and their paths

Flow	Split Node	Split Ratios	
Sunnyvale → New York	Sunnyvale	Los Angeles: Denver:	1/3 2/3
Seattle → Atlanta	Seattle	Denver: Sunnyvale:	1/2 1/2
	Kansas City	Indianapolis: Houston:	1/2 1/2

(c) Splitting ratios for the two flows

Fig. 1: Example network topology with link utilizations

2.1 Network, Paths and Flows

We model a network as a directed simple graph $N = (V, E)$ where V is a finite set of *nodes* (routers) and $E \subseteq V \times V$ is a finite set of *links*. A (simple) *path* in the network is a sequence of distinct nodes $p = v_1 v_2 \dots v_n$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. The total *link utilization* is a function $U : E \to \mathbb{N}$ that assigns to each link its current load.

A traffic *flow* inside the network is a pair $f = (s, t) \in V \times V$ of ingress and egress router, respectively. The *traffic matrix* of the network is a function $F : V \times V \to \mathbb{N}$ such that $F(f)$ where $f = (s, t)$ indicates the amount of traffic that ingress the network at s and egress at t, averaged over some time interval. The set of *paths* used by a flow (from ingress to egress) is given by the set

$P(f) = \{p_1, \ldots, p_n\}$ where each path $p \in P$ starts at the ingress node s and ends in the egress node t.

From the set of paths $P(f)$, we can construct a directed subgraph $G(f) \subseteq E$ of the network where there is an edge $(u, v) \in G(f)$ if and only if there is a path $p \in P(f)$ which contains uv as a subpath. When the network computes the set of paths using equal-cost multi-path routing (ECMP) [26], the subgraph for every flow $f = (s, t)$ is acyclic and the set of possible paths from s to t through $G(f)$ is exactly $P(f)$.

To avoid making assumptions on symmetries of the load balancing weights, we model them independently for each flow f, so that each node v in the flow graph $G(f)$ has a *splitting ratio* $d_v^f : V \to [0; 1]$ such that $d_v^f(u)$ denotes for each next-hop u the percentage of traffic of the flow f that splits at the node v and follows the link (v, u). The flow splitting ratios must satisfy $\sum_{u \in V} d_v^f(u) = 1$ and $d_v^f(u) > 0$ only if $(v, u) \in G(f)$.

Now the fraction of traffic from a flow f on a link e can be calculated using the paths and flow splitting ratios as follows:

$$x_e^f \triangleq \sum_{p \in P(f)} \begin{cases} \prod_{i=1}^{j} d_{v_i}^f(v_{i+1}) & \text{if } p = v_1 \ldots v_n \text{ and} \\ & \quad e = (v_j, v_{j+1}) \\ 0 & \text{otherwise .} \end{cases}$$

The value of x_e^f is the sum the traffic for the flow f over the paths that go through e and for each path we multiply the splitting ratios up until reaching the link e. In the running example in Figure 1, we have for example (ignoring the nodes with no splitting): $x_{hk}^{a \to k} = d_a^{a \to k}(b) + d_a^{a \to k}(d) \cdot d_e^{a \to k}(h) = 1/2 + 1/2 \cdot 1/2 = 3/4$. This means that $3/4$ of the flow size from Seattle to Atlanta passes through the link between Houston and Atlanta.

2.2 Correlation of Traffic Flow and Link Utilization

By correlating the projection of flow traffic onto the links with the actual link utilization U, we can evaluate various hypotheses about the flow splitting ratios, and in that way improve the accuracy of the forward projection of traffic flow.

In an ideal world, we wish to find flow splitting ratios such that the projected traffic matches the actual link utilization:

$$\forall e \in E. \quad U(e) = \sum_{f \in V \times V} F(f) \cdot x_e^f \qquad (ideal)$$

However, due to the inaccuracies of data introduced by e.g. sampling, timing and delay, misclassification or loss of measurements, we cannot expect the projected flow traffic to exactly match the link utilization. Instead, we define

a cost function of how badly the projected traffic on the links differs from the actual link utilization.

$$cost \triangleq \sum_{e \in E} penalty\Big(U(e), \sum_{f \in V \times V} F(f) \cdot x_e^f\Big)$$

where the $penalty$ function describes how undesirable an estimation (est) is given the actual value ($util$), e.g. the absolute error $penalty_{abs}(util, est) \triangleq |util - est|$, or squared relative error $penalty_{rel2}(util, est) \triangleq \left(\frac{util-est}{util}\right)^2$.

Remark 1. In practice, there is a large variety in the size and utilization of links across a network, so $penalty_{abs}$ tends to overfit the large links. Using relative errors alleviates this problem, and squaring the error, like $penalty_{rel2}$ does, penalizes large errors more than several small errors, hence preferring to spread out the inaccuracies over the network.

In the case study, we know that there is some missing data for the traffic matrix F, making it unavoidable that some estimates become too low, so we decide to only penalize over-estimations. Further, in practice the small flows and links are given less importance, so we want to avoid noise in the data with low magnitude having too big an impact on the relative errors. For this we introduce a constant c that is the acceptable absolute error (e.g. $c = 100$Mbps), and arrive at the following penalty function:

$$penalty(util, est) \triangleq \begin{cases} \left(\dfrac{est - util - c}{util}\right)^2 & \text{if } est - util > c \\ 0 & \text{otherwise .} \end{cases} \tag{1}$$

The *flow splitting ratios* (FSR) problem is now to find the splitting ratios d_v^f that minimize the cost function from Equation 1.

3 Solution to Flow Splitting Ratio Synthesis

To solve the FSR problem, we turn to mathematical optimization. In particular, we first encode the FSR problem as the problem of minimizing a linear or quadratic optimization function (depending on the penalty function used) on continuous variables subject to a set of linear constraints. We then study the influence of measurement noise on the precision of splitting ratio estimates.

3.1 Encoding of FSR to a Linear Program

Linear programming (LP) and quadratic programming are well-studied problems with several industry-standard solvers [27, 28]. A linear programming problem is to find values for a vector of decision variables x that minimize a

Define non-negative variables:
$x_{bc}^{b\to i}, x_{bd}^{b\to i}, x_{ab}^{a\to k}, x_{ad}^{a\to k}, x_{eg}^{a\to k}, x_{eh}^{a\to k}, x_{hk}^{a\to k}, err_{ab}, err_{ad}, err_{bc}, err_{bd}, err_{ch}, err_{de}, err_{eg}, err_{eh}, err_{fi}, err_{gf}, err_{gk}, err_{hk}, err_{ji}, err_{kj}$

Minimize:
$(err_{ab})^2 + (err_{ad})^2 + (err_{bc})^2 + (err_{bd})^2 + (err_{ch})^2 + (err_{de})^2 + (err_{eg})^2 + (err_{eh})^2 + (err_{fi})^2 + (err_{gf})^2 + (err_{gk})^2 + (err_{hk})^2 + (err_{ji})^2 + (err_{kj})^2$

Subject to:
(for the flow Sunnyvale $\to$ New York ($b \to i$))
(1) $x_{bc}^{b\to i} + x_{bd}^{b\to i} = 1$
(for the flow Seattle $\to$ Atlanta ($a \to k$))
(2) $x_{ab}^{a\to k} + x_{ad}^{a\to k} = 1$
(3) $x_{ad}^{a\to k} = x_{eg}^{a\to k} + x_{eh}^{a\to k}$
(4) $x_{ab}^{a\to k} + x_{eh}^{a\to k} = x_{hk}^{a\to k}$

(relative link errors)
(5) $err_{ab} \cdot 2 \geq 4 \cdot x_{ab}^{a\to k} - 2 - c$
(6) $err_{ad} \cdot 2 \geq 4 \cdot x_{ad}^{a\to k} - 2 - c$
(7) $err_{bc} \cdot 6 \geq 12 \cdot x_{bc}^{b\to i} + 4 \cdot x_{ab}^{a\to k} - 6 - c$
(8) $err_{bd} \cdot 8 \geq 12 \cdot x_{bd}^{b\to i} - 8 - c$
(9) $err_{ch} \cdot 6 \geq 12 \cdot x_{bc}^{b\to i} + 4 \cdot x_{ab}^{a\to k} - 6 - c$
(10) $err_{de} \cdot 10 \geq 12 \cdot x_{bd}^{b\to i} + 4 \cdot x_{ad}^{a\to k} - 10 - c$
(11) $err_{eg} \cdot 9 \geq 12 \cdot x_{bd}^{b\to i} + 4 \cdot x_{eg}^{a\to k} - 9 - c$
(12) $err_{eh} \cdot 1 \geq 4 \cdot x_{eh}^{a\to k} - 1 - c$
(13) $err_{fi} \cdot 8 \geq 12 \cdot x_{bd}^{b\to i} - 8 - c$
(14) $err_{gf} \cdot 8 \geq 12 \cdot x_{bd}^{b\to i} - 8 - c$
(15) $err_{gk} \cdot 1 \geq 4 \cdot x_{eg}^{a\to k} - 1 - c$
(16) $err_{hk} \cdot 7 \geq 12 \cdot x_{bc}^{b\to i} + 4 \cdot x_{hk}^{a\to k} - 7 - c$
(17) $err_{ji} \cdot 4 \geq 12 \cdot x_{bc}^{b\to i} - 4 - c$
(18) $err_{kj} \cdot 4 \geq 12 \cdot x_{bc}^{b\to i} - 4 - c$
(19) $c = 0$

Fig. 2: Quadratic programming formulation of the example

given cost function $c^T x$ subject to linear constraints $Ax \geq b$ and $x \geq 0$ for some constant vectors b, c and an integer matrix A. In quadratic programming, the cost function can include products of pairs of decision variables, in general: minimize $c^T x + 1/2 \cdot x^T Q x$ for some symmetric matrix Q. We refer to [29] for further introduction to linear and quadratic programming.

In order to describe the encoding of the FSR problem into LP, we need to introduce some notation. Let $G(f)_v^+ = \{(v,u) \in G(f)\}$ be the outgoing edges from the node v in $G(f)$ and let $G(f)_v^- = \{(u,v) \in G(f)\}$ be the incoming edges to v in $G(f)$. The variables of the optimization problem are x_e^f for every flow f and every link e. The value of the variable x_e^f, $0 \leq x_e^f \leq 1$, expresses the fraction of the traffic of the flow f that is traversing the link e. From the x_e^f variables, we can derive the flow splitting ratios as follows

$$d_v^f(u) = \frac{x_{(v,u)}^f}{\sum_{e \in G(f)_v^+} x_e^f}$$

where $d_v^f(u)$ expresses the flow splitting ratio at node v for the next-hop u. The linearly constrained optimization program is then:

minimize $\sum_{e \in E} penalty\big(U(e),\ \sum_{f \in V \times V} F(f) \cdot x_e^f\big)$
subject to $\forall f \in V \times V : \quad$ (let $f = (s,t)$)
(1) $x_e^f \geq 0 \quad \forall e \in E$
(2) $\sum_{e \in G(f)_s^+} x_e^f = 1$
(3) $\sum_{e \in G(f)_v^-} x_e^f = \sum_{e \in G(f)_v^+} x_e^f \qquad \forall v \in V \setminus \{s,t\}$

	t=1	t=2	t=3
flow size $a \to k$	4.2 / 4.0	8.8 / 8.0	18.0 / 6.0
flow size $b \to i$	11.4 / 12.0	22.8 / 24.0	17.1 / 18.0
ratio $d_a^{a\to k}(b)$	51% / 50%	52% / 50%	66% / 50%
ratio $d_a^{a\to k}(d)$	49% / 50%	48% / 50%	34% / 50%
ratio $d_e^{a\to k}(g)$	50% / 50%	50% / 50%	50% / 50%
ratio $d_e^{a\to k}(h)$	50% / 50%	50% / 50%	50% / 50%
ratio $d_b^{b\to i}(c)$	32% / 33%	31% / 33%	6% / 33%
ratio $d_b^{b\to i}(d)$	68% / 67%	69% / 67%	94% / 67%
mean error	0.83%	1.39%	14.36%
max error	1.30%	2.27%	26.99%
link util. ab	2.1 / 2.0	4.6 / 4.0	11.8 / 3.0
link util. ad	2.1 / 2.0	4.2 / 4.0	6.2 / 3.0
link util. bc	5.8 / 6.0	11.8 / 12.0	12.9 / 9.0
link util. bd	7.7 / 8.0	15.6 / 16.0	16.0 / 12.0
link util. ch	5.8 / 6.0	11.8 / 12.0	12.9 / 9.0
link util. de	9.8 / 10.0	19.8 / 20.0	22.2 / 15.0
link util. eg	8.8 / 9.0	17.7 / 18.0	19.1 / 13.5
link util. eh	1.0 / 1.0	2.1 / 2.0	3.1 / 1.5
link util. fi	7.7 / 8.0	15.6 / 16.0	16.0 / 12.0
link util. gf	7.7 / 8.0	15.6 / 16.0	16.0 / 12.0
link util. gk	1.0 / 1.0	2.1 / 2.0	3.1 / 1.5
link util. hk	6.8 / 7.0	13.9 / 14.0	16.0 / 10.5
link util. ji	3.7 / 4.0	7.2 / 8.0	1.1 / 6.0
link util. kj	3.7 / 4.0	7.2 / 8.0	1.1 / 6.0
penalty value	0.008	0.030	13.403

Table 1: Result of quadratic program on three separate simulations where cells show estimated vs. real values

Here we minimize the cost function defined in Section 2.2 and require that (1) the variables a non-negative, (2) the source of the flow initiates all the traffic, and (3) each intermediate router in the flow graph sends out as much traffic as it receives. We do not need a constraint requiring that the target node t receives all the traffic of the flow f, as it is the only sink node in the subgraph $G(f)$, and the constraints (2) and (3) imply that all traffic of f ends in t.

Remark 2. If for some flow f an edge $e = (v, u)$ is the only outgoing edge from v in the subgraph $G(f)$, there is no need to introduce the variable x_e^f as it is redundant.

In order to express the penalty function from the case study (Equation 1), we introduce variables err_e for each link e and rewrite the quadratic program as:

$$\begin{aligned} &\text{minimize} && \textstyle\sum_{e\in E}(err_e)^2 \\ &\text{subject to} && \text{(1) - (3) and } \forall e \in E : \\ &\text{(4)} && err_e \cdot U(e) \geq \textstyle\sum_{f\in V\times V} F(f) \cdot x_e^f - U(e) - c \end{aligned}$$

where the optimal value of the variable err_e is the positive relative error of

	combining time series			*filtering*		
	t=1	t=2	t=3	t=1	t=2	t=3
flow size $a \to k$ (measured / real)	4.2 / 4.0	8.8 / 8.0	18.0 / 6.0	4.2 / 4.0	8.8 / 8.0	18.0 / 6.0
flow size $b \to i$ (measured / real)	11.4 / 12.0	22.8 / 24.0	17.1 / 18.0	11.4 / 12.0	22.8 / 24.0	17.1 / 18.0
ratio $d_a^{a\to k}(b)$ (estimated / real)		64% / 50%			53% / 50%	
ratio $d_a^{a\to k}(d)$ (estimated / real)		36% / 50%			47% / 50%	
ratio $d_e^{a\to k}(g)$ (estimated / real)		51% / 50%			49% / 50%	
ratio $d_e^{a\to k}(h)$ (estimated / real)		49% / 50%			51% / 50%	
ratio $d_b^{b\to i}(c)$ (estimated / real)		18% / 33%			33% / 33%	
ratio $d_b^{b\to i}(d)$ (estimated / real)		82% / 67%			67% / 67%	
mean error		10.01%			1.41%	
max error		15.41%			3.16%	
link util. ab (estimated / real)	2.7 / 2.0	5.6 / 4.0	11.5 / 3.0	2.2 / 2.0	4.7 / 4.0	9.6 / 3.0
link util. ad (estimated / real)	1.5 / 2.0	3.2 / 4.0	6.5 / 3.0	2.0 / 2.0	4.1 / 4.0	8.4 / 3.0
link util. bc (estimated / real)	4.7 / 6.0	9.7 / 12.0	14.6 / 9.0	6.0 / 6.0	12.3 / 12.0	15.3 / 9.0
link util. bd (estimated / real)	9.4 / 8.0	18.7 / 16.0	14.0 / 12.0	7.6 / 8.0	15.2 / 16.0	11.4 / 12.0
link util. ch (estimated / real)	4.7 / 6.0	9.7 / 12.0	14.6 / 9.0	6.0 / 6.0	12.3 / 12.0	15.3 / 9.0
link util. de (estimated / real)	10.9 / 10.0	21.9 / 20.0	20.5 / 15.0	9.6 / 10.0	19.3 / 20.0	19.8 / 15.0
link util. eg (estimated / real)	10.1 / 9.0	20.3 / 18.0	17.3 / 13.5	8.5 / 9.0	17.2 / 18.0	15.5 / 13.5
link util. eh (estimated / real)	0.7 / 1.0	1.6 / 2.0	3.2 / 1.5	1.0 / 1.0	2.1 / 2.0	4.3 / 1.5
link util. fi (estimated / real)	9.4 / 8.0	18.7 / 16.0	14.0 / 12.0	7.6 / 8.0	15.2 / 16.0	11.4 / 12.0
link util. gf (estimated / real)	9.4 / 8.0	18.7 / 16.0	14.0 / 12.0	7.6 / 8.0	15.2 / 16.0	11.4 / 12.0
link util. gk (estimated / real)	0.8 / 1.0	1.6 / 2.0	3.3 / 1.5	1.0 / 1.0	2.0 / 2.0	4.1 / 1.5
link util. hk (estimated / real)	5.5 / 7.0	11.3 / 14.0	17.8 / 10.5	7.1 / 7.0	14.4 / 14.0	19.6 / 10.5
link util. ji (estimated / real)	2.0 / 4.0	4.1 / 8.0	3.1 / 6.0	3.8 / 4.0	7.6 / 8.0	5.7 / 6.0
link util. kj (estimated / real)	2.0 / 4.0	4.1 / 8.0	3.1 / 6.0	3.8 / 4.0	7.6 / 8.0	5.7 / 6.0
penalty value		14.162			0.042	

Table 2: Results of combining the time windows of Table 1 and filtering out the 20% worst constraints (grey cells).

the estimation after discounting the acceptable absolute error c. Note that by using an inequality in the constraint (4), we only penalize over-estimation.

Figure 2 shows the quadratic program for our running example. Here, in case of no measurement noise, the optimal zero-cost solution of the linear program gives us exactly the correct splitting ratios from Figure 1c.

3.2 Measurement Noise

Next, returning to our running example from Figure 1, we synthetically add measurement noise that can vary the size of the measured flow demands. We do this in order to simulate the noise seen in a real network analytics deployment. There is always a small noise variation that reflects the timing and sampling variance, while the large (but less frequent) differences can be caused by late detection of changes in the BGP tables leading to incorrect mapping of ingress-traffic to the flows inside the network.

Table 1 shows the results of three experiments with increasing levels of measurement noise on the first flow (+5%, +10%, +200%), while the second flow has the same small noise (-5%) for all three simulated time windows (t=1,2,3). The measured value is the left number in the cell, and the actual value the right number. Note that, like in real networks, the amount of traffic changes between the time windows. We use the quadratic programming solver CPLEX 22.1 [27] to solve the programs, and report the computed vs. ideal (real) splitting ratios, as well as the forward projected traffic derived from the computed ratios vs. actual utilization on each link based on the real ratios.

As we can see in Table 1, the estimated flow splitting ratios are quite accurate when there is only little noise; however, in the last case (t=3) with a large measurement error for the flow $a \rightarrow k$, the estimated ratios are quite far off. This is even the case for the splitting ratios of the other flow $b \rightarrow i$.

4 Dealing with Measurement Noise

To achieve stable and accurate estimations of the flow splitting ratios despite the noise and occasional large errors in the measurement of the size of the flows, we propose two techniques.

First, by *combining time series* of measurements into a single large quadratic program, we exploit that we have data for multiple time intervals (e.g. 24 one-hour measurements of a day) for which the flow splitting ratios are expected to remain (mostly) unchanged.

Second, by *filtering* out the link error constraints with the highest penalty in the optimization function, we can indirectly filter out the flows with large (but rare) measurement errors. The intuition is that when only a few flows

have large measurement errors, then only relatively few links will be affected. By not considering the utilization of these links for these specific time periods, we effectively filter out the large flow measurement errors without knowing specifically which flows were measured erroneously.

Table 2 shows the result of combining the three time windows from Table 1 of the running example into a single quadratic program. The left side of Table 2 shows the result without filtering, while the right side shows the results of filtering out the 20% link constraints (highlighted with grey background) that contribute the most to the total penalty value of the LP (in the optimal solution for the unfiltered problem).

We see that combining the three time windows reduces the mean error in estimation of splitting ratios from 14.36% for the worst case (t=3) to 10.01% in the combined problem without filtering. After filtering, the mean error is only 1.41%. This small example shows a strong benefit of combining time series of measurements and filtering out some constraints with high penalty; however, it contains only two flows and only one large measurement error. In order to statistically validate the benefits of our technique, we run an extensive simulation experiment on a large set of network topologies.

4.1 Simulation Experiments with Synthetic Traffic

We simulate synthetic flow demands and splitting ratios on real world topologies from the Topology Zoo benchmark [16]. We restrict topologies to their largest connected component (disconnected components can be handled independently) and we do not consider topologies with less than eight nodes or where the synthetic traffic encounters no splitting at all. This leaves us with 192 different topologies.

To generate synthetic traffic, we use the gravity model [30] with random node masses and randomly select 25% of all source-destination pairs to have traffic between them—this corresponds to the numbers found in our industrial case study. As an approximate simulation of the variation of traffic during the day, we vary the total traffic in the network over time using a sine wave together with added noise. We generate 24 traffic matrices, corresponding to one for each hour of the day—a similar setup as the data source in the case study. The splitting ratios are generated by assigning random load balancing weights to the links of the graph and then computing ratios based on these link weights. These demands over time and the splitting ratios are the 'ground truth' of the simulation and are used to compute the true total utilization of each link.

To mimic the type of measurement noise found in a real network analytics deployment, we introduce a small random variation of $\pm 1\%$ to the measured traffic of all flows. We also model rare but large measurement errors in flow traffic: with a low probability of 0.5% we vary a flow size by a random factor between $1/10$ and 10. From the estimated splitting ratios, returned by the

quadratic programing solver, we compute the error compared to the true splitting ratios, and average each of these errors weighted by the total size of the flow. This avoids small, and hence in practice less important, flows dominating and skewing the results. This weighted average splitting ratio error is then considered as the error of that solution.

For each topology, we create ten different random instances of splitting ratios and traffic demands and average the errors. We then report on the best filtering ratio and the error it achieved, and we compare this to the maximum error of a single time window, and to the error when combining time series without filtering. Table 3 orders the topologies by the improvement achieved by filtering compared to only combining time series, and shows the results for the top, middle and bottom seven topologies.

It is clear that combining measurements over multiple time windows balances out the measurement noise and reduces the estimation errors compared to the worst single time window. We can further observe that filtering improves the error in the estimation of the real splitting ratios by an order of magnitude in the best cases and in the middle cases it achieves a significant 66% improvement. In the worst seven topologies, our improvement is smaller (but the filtering technique still improves the precision in every single topology). We observe that topologies with smallest improvement have either close to no error in the first place or have a very large diameter of more than 20, where it is likely to create flows that traverse a large number of links in the network. In such cases, we need to filter up to 50% of LP constraints, which is significantly more than what is needed for the other topologies (where 5-10% filtering is sufficient). Hence if a large measurement error is introduced to such a long elephant flow, then there is simply not enough data on the remaining links in the network to accurately approximate the actual splitting ratios.

5 Scalability Study on Large European ISP

We perform a case study on data from a large European ISP. This network consists of over 3.000 routers and 14.000 links, and the dataset contains hourly traffic matrices, flow paths, and link utilizations for 24 hours of one day. The set of paths used by a flow is in most cases stable in the dataset, but some changes occur during the day. We handle this by assuming that the set of splitting ratios are stable as long as the set of paths is stable, but we introduce new variables for modelling a new set of paths for the flow.

Over the course of one day, more than two and a half million flows have traffic. The quadratic program that analyzes all flows is solved in about seven minutes running on 4 CPU cores at 2.5GHz; however, most of these flows have a very small volume, and in practice the largest flows are the most important. As seen in Figure 3, analyzing the flows that carry 99.9% of the total traffic volume

Topology	Diameter	Max error over all time windows	Error after combining time series	Error after filtering	Filtering percentage	Improvement of filtering
Nextgen	11	40.32%	3.56%	0.20%	5%	94.5%
Bbnplanet	8	56.16%	5.74%	0.43%	5%	92.5%
Psinet	12	50.46%	7.66%	0.64%	5%	91.7%
Goodnet	5	16.36%	2.40%	0.23%	5%	90.5%
Abvt	8	29.62%	4.95%	0.49%	5%	90.1%
Janetlense	5	11.72%	1.69%	0.17%	5%	90.1%
Ibm	7	29.93%	5.46%	0.54%	5%	90.1%
Geant2009	8	22.78%	8.87%	2.87%	10%	67.6%
Geant2001	7	21.48%	10.08%	3.28%	10%	67.5%
Easynet	7	25.87%	4.66%	1.53%	10%	67.1%
Compuserve	5	26.05%	0.82%	0.28%	5%	66.1%
Dfn	7	30.08%	11.41%	3.92%	10%	65.6%
Cesnet201006	7	33.70%	10.65%	3.68%	10%	65.5%
DT	7	19.14%	7.78%	2.71%	10%	65.1%
Ion	26	34.83%	21.13%	15.70%	50%	25.7%
TataNld	29	23.12%	17.05%	13.06%	50%	23.4%
GtsCe	22	28.58%	18.96%	14.59%	50%	23.1%
UsCarrier	36	36.68%	17.00%	13.69%	50%	19.4%
DialtelecomCz	31	34.62%	18.61%	15.17%	45%	18.5%
Gridnet	3	6.16%	0.12%	0.10%	15%	15.7%
Claranet	5	30.28%	0.20%	0.18%	5%	10.4%

Table 3: Experiments with synthetic traffic data

per hour takes only 87 seconds, and analyzing 99% of the traffic volume takes 34 seconds. In conclusion, the method is highly scalable, especially considering the typically uneven distribution of traffic volume in ISP networks.

The diameter of the topology in the case study is 10, so from the experiments on synthetic traffic data we expect that a 5-10% filtering is reasonable. We compute flow splitting ratios using our approach, and observe that they closely match splitting ratios based on link capacities—an insight that is now used in the traffic analytics deployment.

6 Conclusion

We suggested a method for synthesis of flow splitting ratios from incomplete and noisy network traffic flow measurements. Our methods is based on quadratic linear programming and we documented the accuracy and robustness of our method on an extensive synthetic benchmark. Our method is scalable even to large ISP networks. Based on the analysis by our tool on a case

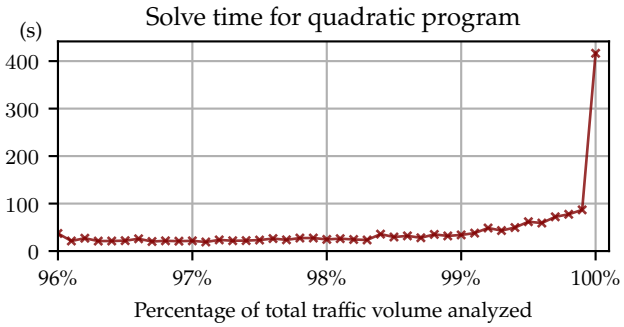


Fig. 3: Scalability of solving FSR on a real, large ISP

study in collaboration with a network analytics company, flow splitting ratios based on link capacities are now used to improve the accuracy of a real traffic analytics deployment.

References

[1] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, "A survey of network flow applications," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567–581, 2013.

[2] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[3] J. L. Garcia-Dorado, A. Finamore, M. Mellia, M. Meo, and M. Munafo, "Characterization of ISP traffic: Trends, user habits, and access technology impact," *IEEE Transactions on Network and Service Management*, vol. 9, no. 2, pp. 142–155, 2012.

[4] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafo, "Five years at the edge: Watching internet from the ISP network," in *Proc. ACM CoNEXT*, 2018, pp. 1–12.

[5] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis, "The lockdown effect: Implications of the COVID-19 pandemic on internet traffic," in *Proc. ACM Internet Measurement Conference (IMC)*, 2020, pp. 1–18.

[6] B. Claise, "Cisco systems NetFlow services export version 9," RFC 3954, Oct. 2004.

[7] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information," RFC 7011, Sep. 2013.

[8] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," RFC 4271, Jan. 2006.

[9] ISO, "Intermediate system to intermediate system intra-domain routeing exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)," ISO/IEC 10589:2002, Nov. 2002.

[10] R. Callon, "Use of OSI IS-IS for routing in TCP/IP and dual environments," RFC 1195, Dec. 1990.

[11] J. Moy, "OSPF version 2," RFC 2328, Apr. 1998.

[12] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional IP routing protocols," *IEEE Communications Magazine*, vol. 40, no. 10, pp. 118–124, 2002.

[13] Juniper Networks, "IS-IS user guide: Understanding weighted ECMP traffic distribution on one-hop IS-IS neighbors," Jan. 2021. [Online]. Available: https://www.juniper.net/documentation/us/en/software/junos/is-is/topics/concept/wecmp-for-one-hop-isis-neighbors-overview.html

[14] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple network management protocol (SNMP)," RFC 1157, May 1990.

[15] D. Harrington, B. Wijnen, and R. Presuhn, "An architecture for describing simple network management protocol (SNMP) management frameworks," RFC 3411, Dec. 2002.

[16] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[17] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, "R3: Resilient routing reconfiguration," in *Proc. ACM SIGCOMM*, 2010, pp. 291–302.

[18] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 97–108, Jun. 2011.

[19] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[20] J. Bogle, N. Bhatia, M. Ghobadi, I. Menache, N. Bjørner, A. Valadarsky, and M. Schapira, "TEAVAR: Striking the right utilization-availability balance in WAN traffic engineering," in *Proc. ACM SIGCOMM*, 2019, pp. 29–43.

[21] K. Lakkaraju, W. Yurcik, and A. J. Lee, "NVisionIP: Netflow visualizations of system state for security situational awareness," in *Proc. Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC '04)*. ACM, 2004, pp. 65–72.

[22] S. J. Saidi, A. Maghsoudlou, D. Foucard, G. Smaragdakis, I. Poese, and A. Feldmann, "Exploring network-wide flow data with Flowyager," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 1988–2006, 2020.

[23] T. Tremel, J. Kögel, F. Jauernig, S. Meier, D. Thom, F. Becker, C. Müller, and S. Koch, "VITALflow: Visual interactive traffic analysis with netflow," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2022, pp. 1–6.

[24] D. Phan, J. Gerth, M. Lee, A. Paepcke, and T. Winograd, "Visual analysis of network flow data with timelines and event plots," in *Proc. Workshop on Visualization for Computer Security (VizSEC 2007)*, ser. MATHVISUAL. Springer, 2008, pp. 85–99.

[25] J. R. Goodall and D. R. Tesone, "Visual analytics for network flow analysis," in *Cybersecurity Applications & Technology Conference for Homeland Security*, 2009, pp. 199–204.

[26] C. Hopps, "Analysis of an equal-cost multi-path algorithm," RFC 2992, Nov. 2000.

[27] IBM, "IBM ILOG CPLEX optimization studio 22.1.0." [Online]. Available: https://www.ibm.com/docs/en/icos/22.1.0

[28] Gurobi Optimization, *Gurobi optimizer reference manual - Version 10.0*, 2023. [Online]. Available: https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/10.0/refman.pdf

[29] R. J. Vanderbei, *Linear programming: Foundations and Extensions*, 5th ed. Springer, 2020.

[30] M. Roughan, "Simplifying the synthesis of internet traffic matrices," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, pp. 93–96, Oct. 2005.

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-659-1

AALBORG UNIVERSITY PRESS