DISS. ETH NO. 31417

Extending the Limits of Formal Methods in Network Operations

Tibor Jan Schneider

DISS. ETH NO. 31417

EXTENDING THE LIMITS OF FORMAL METHODS IN NETWORK OPERATIONS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

TIBOR JAN SCHNEIDER

born on 06.09.1994

accepted on the recommendation of

Prof. Dr. Laurent Vanbever
Dr. Ryan Beckett
Prof. Dr. Todd Millstein

2025

Extending the Limits of Formal Methods in Network Operations
© 2025 Tibor Jan Schneider

Diss. ETH No. 31417

TIK-Schriftenreihe-Nr. 225

Abstract

The Internet has become an essential part of today's society. Critical systems like banking or emergency services depend on the Internet to be highly available. Despite network operators designing highly resilient networks, outages remain a common occurrence, especially those triggered by *human errors*. Indeed, networks have grown in size and complexity, making network operations notoriously challenging.

Academia has proposed systems that apply formal methods to assist in network operations and prevent misconfigurations *before* they are deployed, but the industry has been hesitant to adopt them thus far. We attribute the slow adoption to missing capabilities of these formal tools, as they lack support for critical use cases, network properties, and routing protocols.

This dissertation extends the limits of formal methods in network operations. We present four distinct works that tackle key limitations of existing network verifiers, configuration synthesizers, and reconfiguration systems by extending their capabilities and bringing formal methods in network operations closer to industry-wide adoption.

First, we present the *BGP State Iterator* to explore the space of control-plane routing states directly. Its exploration can be guided by the specification or user-provided heuristics, enabling new use cases such as probabilistic control-plane verification. Second, we present *Velo*, the first verifier to reason about worst-case link loads under unexpected failures *and* routing events. We find that routing events significantly impact link loads in the network, and we design an algorithm to navigate the space of all such events efficiently. Third, we present *Chameleon* to safely reconfigure BGP networks while guaranteeing the absence of routing anomalies at any time, including any transient states, without noticeable overhead. Finally, we present a systematic analysis of the complexity of configuration synthesis. Indeed, we confirm that synthesis is intractable in general, but we also outline a path towards scalable configuration synthesis.

Zusammenfassung

Das Internet ist ein zentraler Bestandteil unserer Gesellschaft. Entscheidende Systeme wie das Bankwesen und Notfalldienste verlassen sich auf die hohe Verfügbarkeit des Internets. Trotz robuster Infrastruktur treten Internetausfälle häufig auf und werden meist durch menschliche Fehler ausgelöst. Netzwerke sind in Grösse und Komplexität signifikant gewachsen, was zu grossen Herausforderungen in deren Betrieb führt.

Systeme aus der Akademie nutzen formelle Methoden, um Netzbetreiber vor Fehlkonfigurationen zu bewahren, *bevor* sich diese negativ auf das Netzwerk auswirken. Allerdings hat die Branche bislang gezögert, solche Systeme zu verwenden. Wir erklären die langsame Einführung dieser formellen Methoden anhand deren fehlender Unterstützung für essenzielle Anwendungen, Netzwerkeigenschaften, und Routingprotokolle.

Diese Dissertation erweitert die Grenzen von formellen Methoden, angewendet im Netzwerkbetrieb. Wir präsentieren vier Forschungsarbeiten, um Limitierungen von Verifikationssystemen, Konfigurationsgeneratoren und Migrationssystemen anzugehen, und formelle Methoden im Netzwerkbetrieb näher zu einer breiten Umsetzung zu führen.

Erstens stellen wir den *BGP State Iterator* vor, welcher alle möglichen Netzwerkzustände findet. Unser Ansatz ermöglicht die geführte Exploration anhand einer Spezifikation und schafft neue Möglichkeiten, wie die wahrscheinlichkeitstheoretische Netzwerkverifikation. Zweitens präsentieren wir *Velo*, das erste System, um die schlimmstmögliche Netzauslastung aufgrund unerwarteter Ausfälle und Routing-Ereignisse zu finden. Wir demonstrieren den starken Einfluss von Routing-Ereignissen auf die Netzauslastung und zeigen eine Methode zur effizienten Navigation des gesamten Ereignisraums. Drittens stellen wir *Chameleon* vor, um BGP-Netzwerke sicher zu migrieren. Als erstes System können wir garantieren, dass die Spezifikation zu jedem Zeitpunkt eingehalten wird. Zuletzt präsentieren wir eine systematische Analyse der Komplexität der Konfigurationssynthese. Wir bestätigen, dass die Synthese im Allgemeinen nicht in polynomischer Zeit lösbar ist, skizzieren aber auch einen Weg zu skalierbarer Konfigurationssynthese.

Acknowledgments

After writing several highly technical papers and enjoying this process, I find myself struggling to write this section. Not only is technical writing more natural for me, but I also never paused to contextualized the four years of my doctorate. Looking back, I truly enjoyed my entire journey; it was very demanding, but also exciting, (mostly) fun and surprisingly smooth. Yet, observing colleges dealing with setbacks from technical hurdles and periods of exhaustion often stirred in me a sense of guilt that is difficult to articulate. I often say that my doctorate was smooth due to luck—luck that my initial endeavors turned out to work sufficiently and luck that my work was well received by the research community. While that is most certainly true to some degree, I realize now that people surrounding me had a far greater impact on my success. By not being afraid to ask the difficult questions, they helped me to avoid many pitfalls and dead ends. I am profoundly grateful for the guidance, support, and collaboration of so many extraordinarily gifted people.

First and foremost, I want to express my utmost gratitude to Laurent Vanbever. Your rigorous, critical questioning fundamentally shaped my ideas into the structured form they now take. Besides demonstrating the art of thinking, you taught me the importance of communicating ideas in terms of writing papers and in giving talks in conferences and lectures. Further, I deeply appreciate your trust in my judgment, allowing me to explore directions that I considered promising.

After all, your engaging lectures were the main reason for doing projects and starting my PhD in your group.

Likewise, I extend my sincere thanks to Stefano Vissicchio. Your deep technical insights greatly enriched this dissertation, and your unfailing determination to seeing our projects to the finish line have been a continuous source of motivation for me. Further, I am grateful to João Luís Sobrinho for your technical comments and suggestions, and Simon Leinen for providing anonymized and aggregated traffic data of Switch.

Next, I want to thank Todd Millstein and Ryan Beckett for serving as co-examiners for this dissertation. They have shaped this dissertation directly by their insightful comments, and indirectly by their seminal publications in the field of network verification and synthesis.

I am deeply grateful to the people at Google who made my internship such a memorable experience. In particular, I want to thank Anees Shaikh and Xuqian Ma for their guidance, encouragement, and for giving me the freedom to steer the project and explore adjacent ideas.

I also want to thank all members of the Networked Systems Group for making the time during my PhD joyful. I will fondly remember our retreats, group activities, and the day-to-day discussions; no topic too far-fetched, and no premise too absurd. I wish the relationships we built lasts for many years to come. In particular, my heartfelt thanks goes to my office mate Roland. Our tight collaboration—and your refreshingly direct way of telling me when an idea was terrible—kept improving my work. You helped me far more than you realize.

Finally, I owe my deepest gratitude to my friends and family. Thank you for graciously enduring my stressful periods, always offering an escape from the lab, and bravely listeningto research ideas that you couldn't possibly understand. Your constant love was the stability I needed for this immense undertaking, and I dedicate this dissertation to you.

Naming our systems after animals made them only marginally more exciting to a layman.

Tibor, November 2025

Publications

This dissertation is based on papers published in conference proceedings presented hereafter.

On the Complexity of Network-Wide Configuration Synthesis
Tibor Schneider, Roland Schmid, Laurent Vanbever
Proceedings of the 30th International Conference on Network Protocols (ICNP).
Lexington, KY, USA, November 2022.

Taming the transient while reconfiguring BGP
Tibor Schneider, Roland Schmid, Stefano Vissicchio, Laurent Vanbever
In *Proceedings of the 37th Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*
New York, NY, USA, September 2023.

Verifying maximum link loads in a changing world
Tibor Schneider, Stefano Vissicchio, Laurent Vanbever
In *Proceedings of the 22th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
Philadelphia, PA, USA, April 2025.

Guided Exploration of Control-Plane Routing States
Tibor Schneider, Jean Mégret, Laurent Vanbever
In *Proceedings of the 33th International Conference on Network Protocols (ICNP).*
Seol, South Korea, September 2025.

The following publications were part of my PhD, but they were led by other researchers and only referenced in this dissertation.

Predicting Specification Violations During BGP Convergence

Roland Schmid, Tibor Schneider, Georgia Fragkouli, Laurent Vanbever

In *Proceedings of the Student Workshop at the 19th International Conference on emerging Networking EXperiments and Technologies (CoNEXT-SW).*

Paris, France, December 2023.

Causality Analysis in Control Plane Verification

Yu Chen, Tibor Schneider, Laurent Vanbever

In *Proceedings of the Student Workshop at the 19th International Conference on emerging Networking EXperiments and Technologies (CoNEXT-SW).*

Paris, France, December 2023.

The Effects of iBGP Convergence

Roland Schmid, Tibor Schneider, Georgia Fragkouli, Laurent Vanbever

Technical report,, March 2025.

Transient Forwarding Anomalies and How to Find Them

Roland Schmid, Tibor Schneider, Georgia Fragkouli, Laurent Vanbever

In *Proceedings of the 21st International Conference on emerging Networking EXperiments and Technologies (CoNEXT).*

Hong Kong SAR, China, December 2025.

Contents

1 *Introduction* 1

2 *Background and Related Work* 5
- 2.1 Internet Routing 5
- 2.2 Formal Methods for Network Management . . 6
- 2.3 Models for Routing Protocols 11
- 2.4 An Efficient BGP Network Simulator 12

3 *Guided Exploration of Routing States* 16
- 3.1 Overview 18
- 3.2 Model of BGP 22
- 3.3 Backtracking Algorithm 24
- 3.4 Supported Specifications 29
- 3.5 Evaluation 31
- 3.6 Conclusion 38

4 *Verifying Maximum Link Loads in a Changing World* 39
- 4.1 Overview of *Velo* 42
- 4.2 Model and Notation 47
- 4.3 Finding Maximum Link Loads 48
- 4.4 Approximating the Traffic Matrix 56
- 4.5 Evaluation 60
- 4.6 Case Study 68
- 4.7 Related Work 70
- 4.8 Discussion 71
- 4.9 Conclusions 72

5 *Taming the transient while reconfiguring BGP* 74

5.1 Overview 77

5.2 Analyzer 82

5.3 Scheduler 85

5.4 Compiler 89

5.5 Case Study 92

5.6 Evaluation 93

5.7 Limitations and Discussion 101

5.8 Related Work 104

5.9 Conclusion 105

6 *On The Complexity of Configuration Synthesis* 106

6.1 Model 108

6.2 Complexity Analysis 116

6.3 Related Work 128

6.4 Conclusion 129

7 *Conclusion and Outlook* 131

7.1 Open Problems 132

Bibliography 138

1 Introduction

The Internet is one of the most influential technical advances in human history thus far, enabling the global exchange of ideas and information, on-demand video streaming, and convenient payment systems, to name a few. We rely on the Internet for almost everything nowadays without even realizing it. This dependency only becomes apparent once something fails—which, unfortunately, happens (too) frequently.

There are numerous reports of severe outages in Internet Service Providers (ISPs) and enterprise networks [11–13]. For example, consider the Meta (then Facebook) outage in October 2021 that affected Facebook, Instagram, and WhatsApp for six hours, causing a revenue loss of 60 million dollars [14]. Besides taking down an entire infrastructure, such failures can impact parts of the Internet outside the responsible company. For instance, in August 2017, Google accidentally disconnected parts of Japan from the rest of the Internet [15]. Unfortunately, given how pervasive the Internet has become, these outages even impact critical emergency services. For example, in March 2021, Swisscom, the national ISP in Switzerland, experienced a short-lived outage that impaired emergency telephone service for 15 minutes [16]. Despite that, just a couple of months later, in July 2021, another outage at Swisscom impaired emergency services, but this time for the entire night [17].

Fortunately, emergency services could redirect some calls to a legacy fallback system.

Despite operators and engineers designing resilient services that can tolerate failures of parts of the physical infrastructure, outages *still* happen. Indeed, the aforementioned outages are caused by *human error*. Some were traced back to bugs in the application layer, while misconfigurations in the network layer caused the others. Network misconfigurations are particularly critical, mainly for two reasons. First, a simple typo can have Internet-wide consequences, for instance, by causing route leaks [18–21]. Second, networks have grown in both their size and complexity, running multiple complex and interconnected distributed routing protocols. Managing such networks has become incredibly challenging.

In the past two decades, both academia and industry have proposed many techniques and systems to reduce the frequency of misconfigurations and outages by using formal methods. We broadly distinguish those systems into three categories. First, *Verifiers* try to prove that a configuration is correct, i.e., whether a network running this configuration satisfies some *network properties* under uncontrollable events like failures and routing events, which we call an environment. If the configuration is invalid, verifiers yield a counterexample, that is, an environment in which the network violates a property. The second category of tools directly *synthesizes* configurations: finding a configuration that satisfies a given specification or automatically fixing the existing, erroneous configuration. The third category aims at safely *reconfiguring* a network, i.e., to transition from an old to a new configuration while satisfying the specification during the entire transition. This includes the time during which the distributed network converges, a chaotic process that is difficult to predict [8, 9].

Indeed, these formal methods promise to catch routing anomalies and prevent outages *before* they occur. Despite major advances in this field and the emergence of powerful tools in all three categories, industry adoption has been significantly slower, and large-scale Internet outages *still* happen [13, 20, 22]. While there are numerous reasons for this slow rate of adoption, we mainly attribute it to missing capabilities that these tools currently offer. This prevents operators from deploying these systems, especially since adding support for lacking capabilities tends to be far from trivial.

State-of-the-art verification, synthesis, and reconfiguration systems indeed lack capabilities in three domains: First, they fail to support key *use cases*. For instance, in case of a violation, verifiers only yield arbitrary counterexamples, neglecting their relevance and ignoring operators' domain knowledge of likely future routing events and failures. Second, they often fall short in which *network properties* they can reason about. Specifically, performance-related ones like link loads are very insightful for operators but are only supported in a very rudimentary form. Third, they usually cannot generalize to other *routing protocols*. Reconfiguration tools especially apply only to simple reconfigurations for which their convergence process is well understood. Unfortunately, this excludes the Border Gateway Protocol (BGP), which forms the backbone of today's Internet.

In general, existing systems are limited by their underlying formal methods like Satisfiability Modulo Theories (SMT). Supporting new use cases, network properties, or routing protocols tends to be either inconsistent with their model or lead to an explosion of the search space.

This dissertation extends the limits of formal methods in network operations to support additional use cases, network properties, and protocols: The *BGP State Iterator* [1] enables new *use cases* by finding the relevant counterexamples and paves the way towards probabilistic control-plane verification. *Velo* [2] extends control-plane verification to new performance-related *properties* by finding potential future congestion events. *Chameleon* [3] enables safe and practical reconfigurations of complex interdomain *routing protocols* such as BGP. Finally, our *Complexity Analysis* [4] investigates the tractability of synthesis problems for pairs of *protocols* and *network properties*, laying the foundations for scalable network-wide configuration synthesis. These four distinct works tackle key limitations of existing systems and extend their capabilities, bringing formal methods applied to network operations closer to industry-wide adoption.

The BGP State Iterator. Existing verifiers explore the space of environments, i.e., failures and routing events, to find a counterexample. Yet, operators reason about the network not in terms of environments but its state, such as forwarding and route propagation paths. This semantic gap prevents users from guiding and influencing the verifiers' exploration according to their high-level intent and domain-specific knowledge.

In Chapter 3, we introduce the *BGP State Iterator*, a novel algorithm that directly leverages user-provided specifications, heuristics, or a probabilistic model to find *relevant*, i.e., likely, counterexamples *quicker* without exploring the entire space. This enables verifying the probabilistic nature of Service-Level Agreements, e.g., providing connectivity 99.99% of the time.

Velo. Most control-plane verifiers today focus on functional properties that pertain to forwarding and propagation paths, but they cannot verify properties that relate to performance [23]. Indeed, reasoning about such properties requires scrutinizing packet forwarding for all destinations simultaneously, while existing verifiers consider one destination (or a few) at a time.

In Chapter 4, we present *Velo*, the first system to find worst-case link loads under a wide range of network environments. Specifically, link loads are affected not only by failures but also by routing events that are triggered externally but still change forwarding paths inside the network. Instead of exploring all possible routing events, we prove that considering only a few events is sufficient to find worst-case link loads.

The few existing systems capable of reasoning about performance only consider failures, but they ignore external routing events.

Chameleon. Upon reconfiguring a network, routers exchange routing information to converge to a new state. However, this convergence process often results in transient forwarding anomalies, such as blackholes and forwarding loops, even if both the initial and final configurations are correct [9]. Existing reconfiguration systems can avoid such anomalies. However, they are impractical by imposing significant overhead [24, 25] or can only guarantee safety for a small subset of migrations and protocols [5, 26, 27], such as changing link weights.

In Chapter 5, we introduce *Chameleon*, the first system to reconfigure BGP networks safely and practically. We achieve this by finding one convergence process free of anomalies and implementing it using temporary configuration commands. We formulate the reconfiguration as a scheduling problem and rely on Integer Linear Programming (ILP) to plan and apply large-scale reconfigurations within minutes.

The complexity of synthesis. Configuration synthesis avoids the error-prone task of manually writing configuration files. Instead, operators specify a high-level intent; the synthesis system then automatically finds a matching configuration or updates an existing one. While promising, existing systems today either synthesize one specific aspect of the configuration or struggle at scaling to realistic networks when given realistic properties to satisfy [28–31].

In Chapter 6, we systematically analyze the computational complexity of configuration synthesis. We focus on different routing protocols to implement numerous types of network properties and study necessary sub-problems to synthesize configurations. Perhaps unsurprisingly, configuration synthesis is (at least) *NP*-complete in general. However, we show specific combinations of routing protocols and network properties that are solvable in polynomial time and outline a path towards scalable configuration synthesis.

Before diving into our technical contributions, we provide the necessary background in Chapter 2. We present and discuss related work in of modeling routing protocols and verifying or synthesizing their configurations. We also present *BGPSim*, a control-plane simulator that proved vital during our research endeavors. Finally, in Chapter 7, we discuss yet open research problems to conclude this dissertation.

2 Background and Related Work

In this chapter, we provide the necessary background for this dissertation. Section 2.1 introduces Internet routing including common interdomain and intradomain routing protocols. In Section 2.2, we highlight existing network management tools and summarizes related works. Next, Section 2.3 describes two commonly used models to capture routing protocols. Finally, Section 2.4 describes *BGPSim*, a network simulator we develop to support our research in this dissertation.

2.1 Internet Routing

The Internet is a network of networks. Over 75 000 independent networks [32] called Autonomous Systems (ASes) establish commercial relationships and exchange Internet traffic. Internet routing is the process by which IP packets delivered to their destination. Distributed routing protocols ensure packets eventually reach their destination despite the dynamic nature of the Internet and the reluctance of ASes to share details about their networks and operations. We distinguish *intradomain* and *interdomain* routing. The former describes how packets are forwarded within an AS, while the latter governs how packets are routed between ASes.

Interdomain routing. Interior Gateway Protocols (IGPs) find optimal paths *inside* an AS. Link-state protocols like Open Shortest Path First (OSPF) [33] and Intermediate System to Intermediate System (IS-IS) [34] distribute the network-wide topology to each router, so each router can compute its optimal paths to all destinations within the AS. In contrast, distance-vector protocols such as EIGRP [35] only exchange distances to destinations with neighbors, thus computing optimal paths in a distributed manner.

Optimality can refer to delay, i.e., finding shortest paths, or throughput, i.e., widest paths.

Centralized approaches, i.e., Software Defined Networking (SDN), enable flexible traffic engineering by computing paths centrally and directly installing the forwarding tables[36].

Intradomain routing. Internet-wide routing is achieved using the Border Gateway Protocol (BGP) [37], a path-vector protocol that distributes paths and a set of other attributes to remote destinations. ASes choose and distribute paths according to monetary relationships with neighboring networks. Indeed, all interdomain links in the Internet are commercial by nature. Either *(i)* one AS "provides" Internet access to a customer, or *(ii)* both parties see mutual benefit in exchanging traffic, so both just pay for maintaining the necessary physical infrastructure.

BGP is designed to be extremely flexible and accommodates such (and other) commercial relationships. One common way to implement these commercial relationships is using Gao-Rexford routing policies [38]. Indeed, the *local-preference* attribute of a route is used to prefer routes from a customer over those from a provider. Likewise, provider routes are tagged using the *community-set* to prevent them from being exported to another provider, thus paying twice and receiving nothing in return.

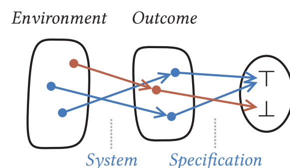
BGP routes are propagated over sessions. We distinguish External BGP (eBGP) sessions that connect two different ASes and Internal BGP (iBGP) sessions to disseminate routes within an AS. By default, a router will never propagate a route learned from an iBGP neighbor to another one, necessitating a full-mesh of iBGP sessions between all pairs of routers. To reduce the overhead associated with each session, operators can configure routers as *Route Reflectors* to distribute routes within an AS, and hence reducing the number of sessions required [39].

2.2 Formal Methods for Network Management

In the past two decades, both academia and industry have proposed tools to help operators safely manage their networks. Many of these tools are *reactive*: they continuously monitor the network and alert the operator upon detecting anomalies or outages [40–43]. Others are *proactive*, relying on formal methods to prevent issues from happening in the first place by *(i) verifying* that the configuration adheres to a given intent, *(ii)* directly *synthesizing* a configuration from the intent, or *(iii) reconfiguring* a network to safely transition from the initial to the final configuration. In the following, we detail each kind of system separately and discuss related works.

2.2.1 Network Verification

Network verifiers take a configuration as input and check that certain properties are satisfied. In their simplest form, network simulators allow operators to perform what-if analysis [44, 45]: How would the network perform if a specific link to a peering network fails? What if a peer starts advertising better routes? In contrast, verifiers prove that the specification is satisfied in *all* such environments. Verifiers target all layers of the network stack, from physical queues [46, 47] over distributed routing protocols [48, 49] and congestion control [50] to the Domain Name System (DNS) [51]. In this dissertation, we focus on verifiers for the network layer, and distinguish them according to what they consider as an *environment* over what kinds of *outcomes* they can reason about (i.e., network properties).



Network verification models the system (network) as a mapping of the environment to an outcome, and the specification defines which outcomes are expected and which are undesired. Verification aims to find an environment that leads to an undesired outcome, i.e., a counterexample shown in red.

Data-plane verifiers. Such systems explore the space of packet headers (the environment) and verify properties relating to how these packets are forwarded and modified (the outcome). They (mostly) treat the control plane as fixed, only considering the generated forwarding rules. Some of these aim at generality by defining abstract languages for representing arbitrary data planes [52–55]. Others focus on off-the-shelf routers and obtain the forwarding tables from the live network or by simulating the control plane [45, 56–58]. The last set of tools focuses on network programmability by verifying P4 data-plane programs or forwarding rules generated by SDN [59, 60].

Control-plane verifiers. Instead of assuming a fixed control-plane, these verifiers reason about how link failures routing events, such as changes in BGP routes received from peering networks (the environment), affect the resulting network (the outcome). While many verifiers support forwarding properties as well, they often focus on routing properties, e.g., whether a provider's route is leaked to another provider. These systems also differ significantly in terms of assumptions and solving techniques and, thus, what kinds of networks they can support.

Systems like *Arc* [61], *Tiramisu* [62], *Plankton* [63], and *SRE* [64] can verify such properties under arbitrary link failures, but they expect a fixed set of BGP routing advertisements as input. In contrast, others like *Bagpipe* [65], *Minesweeper* [48], *NV* [66], *Acorn* [67], and *Expresso* [49] consider failures and arbitrary routing inputs from BGP.

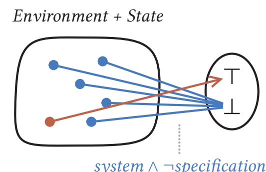
While most of these systems verify routing and forwarding properties, some go further. Systems like *ProbNV* [68] and *NetDice* [23] verify probabilistic Service-Level Agreements (SLAs). Given a probabilistic model over link failures, they check whether a property, say reachability, is satisfied with 99.99% probability. However, none of the existing verifier can reason about the likelihood of different BGP routing events.

Network verifiers typically analyze a single destination at a time and thus can only verify *local* properties such as individual propagation and forwarding paths. *Global* properties that relate to the network's performance, however, depend on the state of all destination prefixes at the same time. *QARC* [69], *YU* [70], and *Raha* [71] verify about worst-case link loads properties under failures. Yet, these systems are limited to link failures and cannot reason about changes in BGP routing inputs.

Verifiers can prove properties for all prefixes at once by representing the prefix as a symbolic variable.

Network verifiers differ significantly in how they explore the space of environments and find counterexamples. Some systems represent all routing protocols and their configurations using distances in an abstract graph, enabling them to rely on common graph algorithms to find counterexamples [61, 62, 69]. In doing so, they limit themselves to a subset of protocols that can be expressed in terms of finding shortest paths, which excludes most BGP networks. Others symbolically simulate the network by propagating symbolic routes that are represented using Binary Decision Diagrams (BDDs) [49, 64]. Doing so allows these systems to return the entire space of environments that violate the specification instead of a single counterexample.

The vast majority of control-plane verifiers rely on SMT solvers [48, 66, 67, 72–75]. These systems represent both the environment and the resulting routing state using symbolic variables and relate them using SMT constraints that describe the network's stable state. To make these systems scale to large networks, researchers have proposed several techniques to essentially reduce the problem size. *Bonsai* [72] proposes to find equivalent routers that can be combined. Others, such as *Lightyear* [73], *Timepiece* [74], and *Kirigami* [75], propose to split up the verification problem by partitioning the topology into smaller components. They define an interface between the components and invoke an SMT solver to verify each individually, thus scaling not in terms of the network size but in the size of the largest component.



SMT-based verifiers capture the relation between the environment and routing state as a set of SMT constraints that describe the system itself. The solver aims at finding an environment and routing state that satisfies those system constraints but violates the specification.

2.2.2 Configuration Synthesis

Network synthesis aims at generating configurations that are provably correct, enabling intent-driven networking [76–78]: Instead of manually writing low-level configurations, operators only express a high-level intent, and the system automatically finds a configuration that implements that intent. Doing so replaces the labor-intensive and error-prone process of writing configuration files but also enables optimizing objectives such as network throughput in traffic engineering systems [79–82]. Such tools can also fix configuration bugs by finding a new configuration while minimizing the difference to the old one.

General configuration synthesis tools such as *CPR* [83], *SyNet* [84], *Zeppelin* [30], *NetComplete* [29], and *AED* [85] aim at finding an OSPF and BGP configuration for a given topology that satisfies the specification. They rely on techniques such as Counterexample-Guided Inductive Synthesis (CEGIS) [86] to explore the space of possible configurations. Yet, doing so simply does not scale to today's networks, as they essentially need to verify the configuration repeatedly.

Protocol-specific synthesis tools such as *Propane/AT* [28, 87] and *Aura* [31] use a fixed set of rules to transform policies expressed in a domain-specific language to a set of BGP configurations. By relying on fixed compilation rules, these tools can scale to large production-scale networks. As we show in Chapter 6, doing so is possible thanks to the flexibility that BGP configurations offer. Interestingly, generating configurations from policies can be computationally easier than verifying that an arbitrary configuration implements these policies. This is because the generated configurations show a lot of structure and symmetry that is not present in arbitrary configurations.

Finally, some synthesis tools rely on recent advances in machine learning, such as Graph Neural Networks and Large Language Models (LLMs), to generate configurations [88–92]. To avoid syntactic and semantic errors, they usually rely on a verifier and feed the output back into the LLM for the next iteration. This still allows them to provide guarantees that their generated configurations are correct. Unfortunately, they fundamentally cannot guarantee that they will eventually find a configuration or that their process will terminate if no correct configuration exists.

2.2.3 Network Reconfiguration

Most routing protocols today are distributed; routers exchange routing information with neighbors and gradually converge to a stable state, a process that we call the convergence of a network. During this convergence process, the routing state of different routers tends to be inconsistent, where one router selects a path even though others along that path prefer a different one. This results in forwarding anomalies, especially with hop-by-hop routing protocols in which each router along the path forwards packets to its next hop, potentially trapping packets in a loop.

Reconfiguring the network usually triggers a convergence process that likely causes such anomalies during convergence regardless of whether both the initial and final configurations are correct. Such anomalies are extremely challenging to avoid, even when following best practices [93]. By relying on formal methods, researchers have proposed techniques to reconfigure networks while maintaining guarantees throughout the entire reconfiguration and the inevitable convergence process.

The strongest consistency guarantees provide systems based on Software Defined Networking (SDN) by directly controlling the forwarding rules [94–96]. Some systems guarantee that each packet is processed entirely by either the old or the new configuration but not by a mixture of the two [97–99]. Others relax those constraints, aiming at satisfying a user-provided specification at each intermediate step [100–103]. However, such approaches require a central controller. In theory, they could be used to reconfigure networks running distributed routing protocols by taking over the forwarding tables using static routes. Doing so, however, would hinder the network's ability to automatically react to failures and routing changes.

To safely reconfigure distributed routing protocols, one approach is to duplicate the entire control plane on all routers to run both the old and the new configuration at the same time [93, 104]. Initially, the old configuration still drives the data plane, allowing the new configuration to converge in the background. These systems then gradually instruct routers to "activate" the new configuration in an order that avoids forwarding loops [24, 25]. While useful, these tools tend not to be used in practice due to their significant overhead—running the new configuration in the background requires duplicating all routing tables that store millions of entries.

Other systems perform the reconfiguration "in-place" by splitting it into small changes and gradually applying them to maintain correctness. Providing strong guarantees while doing so is challenging, as one must consider all possible ways in which the network might converge. Some approaches focus on link weight changes in an IGP [26, 27] and modifying the link weights in small increments. Others, such as *Snowcap* [5] and *Cursor* [105] achieve generality by splitting the configuration update into its smallest units and applying them in an order in which all intermediate configurations are correct. However, these systems neglect any convergence process: they cannot provide strong safety guarantees during convergence.

2.3 Models for Routing Protocols

Routing protocols deployed in the Internet are diverse: whether they are link-state, distance-vector, or path-vector protocols, what metrics they optimize (shortest, widest, or most reliable path), and how many tunable parameters they have. Instead of modeling each protocol individually, the literature has provided tools to explain them using abstract models. In the following, we discuss two of them that are relevant to this dissertation. *Routing Algebra* is commonly used to capture optimal or stable states of a distributed protocol [106]. In contrast, the Simple Path Vector Protocol (SPVP) is an abstract model designed specifically to capture the stability of BGP [107].

Routing Algebra. Any routing protocol tries to find optimal paths according to some attributes. The way these attributes are transformed along the path and how they are compared have significant implications for the protocol's behavior [108–110]. A routing algebra models a routing protocol as a four-tuple $(A, \preccurlyeq, \oplus, \bullet)$ containing attributes A, a comparison relation $\preccurlyeq$ (with $\prec$ as the strict version), and an extension operator $\oplus$. The absorbing attribute $\bullet$ represents the absence of a route.

Shortest-path to minimize latency is modeled as $(\mathbb{R}^+, \leq, +, \infty)$*, and widest-path to maximize available throughput is* $(\mathbb{R}^+, \geq, \min, 0)$

Algebraic properties of the routing algebra imply properties on the stable state of any routing protocol that is computing optimal paths in this way. For instance, if the routing algebra is such that any extension of an attribute makes this attribute less preferred, i.e., attributes always become worse (a property that is called *monotonicity*), then there always exists a stable routing state when implemented using a distance-vector protocol.

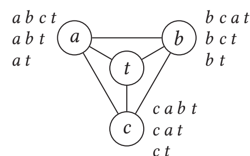
A routing algebra $(A, \preccurlyeq, \oplus, \bullet)$ *is monotone if and only if* $\forall a, b \in A$, $a \preccurlyeq b \oplus a$.

Likewise, if the relation of two attributes remains unchanged when extending by the same third attribute (called *isotonicity*), then any stable routing state implements optimal paths.

A routing algebra $(A, \preccurlyeq, \oplus, \bullet)$ is isotone if and only if $\forall a, b, c \in A$, $a \preccurlyeq b \implies c \oplus a \preccurlyeq c \oplus b$.

The routing algebra of shortest paths is both monotone and isotone. Consequently, protocols like OSPF that find shortest paths will always converge to a stable and optimal state. To analyze non-monotone protocols, the literature proposes to check the properties of cycles in the network [111]. Specifically, the absence of a non-inflationary cycle still ensures that a distance-vector protocol finds a stable state. On the contrary, BGP, in general, is neither monotone nor isotone and often contains non-inflationary cycles, which indicates that it *might* not converge to an optimal state or may not converge at all.

Simple Path Vector Protocol (SPVP). This model represents an abstract version of BGP that operates solely on paths. Each node receives paths from its neighbors, selects one of them according to its *local* preference ranking, and notifies its neighbors once it selects a new route. As each node maintains its own ranking, this protocol is neither monotone nor isotone.



A dispute-wheel has no stable path assignment [107]. No matter which paths are chosen, there is either a more preferred path available or the path is not valid and will be withdrawn.

This model is used to analyze the Stable Path Problem (SPP), that is, whether a stable path assignment exists. In a stable path assignment, each node selects the best *available* path, and a path is available if the first node along the path also selects that path. Most control-plane verifiers today rely on this insight to analyze the network: they only consider stable routing states as a generalization of a stable path assignment for SPVP.

2.4 An Efficient BGP Network Simulator

Network simulation remains an invaluable tool for network operators and researchers [112–114] despite the emergence of verification. For instance, many data-plane verifiers simulate the control-plane to obtain forwarding tables for performing their verification [45, 57, 58]. Our research heavily relies on network simulation for various tasks. *Snowcap* [5] simulates different reconfiguration sequences to find those that comply with the specification. Likewise, *BGPseer* [6] repeatedly simulates the network with different timing models to study anomalies during convergence. *Chameleon* [3] relies on accurate router-internal information to generate reconfiguration plans.

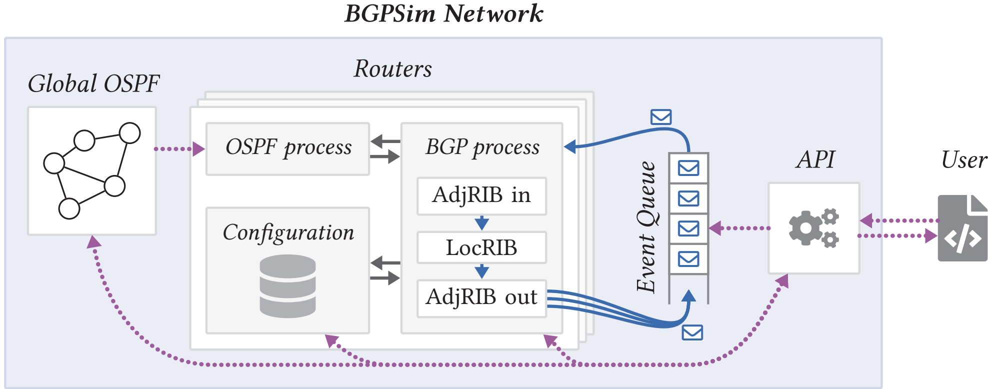


Figure 2.1: Overview of the main components of BGPSim. Blue solid arrows show control-plane events, while purple dotted arrows show the control flow when using BGPSim's API.

BGPSim is implemented in over 40 000 lines of Rust code. It is available under an Apache 2.0 license at github.com/nsg-ethz/bgpsim.

We develop our own BGP network simulator *BGPSim* to meet the requirements of our research. We optimize *BGPSim* for *(i)* performance as *Snowcap* tests millions of update sequences, *(ii)* extensibility for *BGPseer* uses custom timing models, and *(iii)* transparency, exposing router-internal data structures that *Chameleon* needs. We implement *BGPSim* and compare its performance against *C-BGP* [44] and *Batfish* [45]. *BGPSim* is 460× faster than *C-BGP* and 40× faster than *Batfish*.

We develop *Snowcap*, *Chameleon*, *Velo*, and the *BGP State Iterator* around *BGPSim*, relying on its functionality for parsing and preprocessing the configuration and routing state. *BGPSim* provides a simple interface to load topologies from Topology Zoo [115] and automatically configure them with simple but realistic BGP and OSPF configurations for quickly evaluating our systems on different networks and configurations.

2.4.1 Network Model

When designing a network simulator, the chosen network model is critical, as it affects the performance, the expressivity, and the ease of use. Generally speaking, accurately modeling the network improves expressivity at the cost of performance, while an abstract model reduces necessary computations. For teaching purposes, a high-level abstraction best visualizes the protocol's core dynamics, while an accurate network simulator (ideally a virtualized network of real devices) highlights the interactions between protocols and networking concepts.

BGPSim relies on an abstract network model to optimize for performance, similar to the one used in *C-BGP* [44]. It focuses on the selection and propagation of BGP routes, ignoring how control-plane packets traverse the network. The following details how *BGPSim* model the topology, the IGP and BGP.

Intradomain Routing. *BGPSim* simulates OSPF [33]. The user can decide whether *BGPSim* should compute the OSPF routing state globally or by simulating how OSPF exchanges Link State Advertisements (LSAs). In the global mode, *BGPSim* computes the shortest path for all pairs of routers and then distributes the resulting routing tables to all routers. Alternatively, *BGPSim* can simulate the distributed computation of OSPF, where routers exchange LSAs with their neighbors as control plane messages. Doing so allows users to reason about the network while OSPF and BGP both converge simultaneously. As such, *BGPSim* must recompute all shortest paths upon receiving any new LSA.

BGPSim does not simulate the periodic *Hello* packets that are exchanged between neighbors. Instead, upon simulating a failure, it immediately triggers adjacent routers to flood an updated LSA. Likewise, we do not increase the *Age* field of an LSA each second as the specification requires [33] and assume that routers periodically flood updated LSAs. *BGPSim* replicates LSA sequence number overflows, which can cause transient routing anomalies until the old LSA is removed from all routers.

Interdomain Routing. *BGPSim* exchanges BGP control-plane messages to capture BGP's behavior accurately. Each router maintains its Routing Information Base (RIB). Upon receiving BGP *Update* and *Withdraw* messages, the BGP routing process recomputes its RIBs according to the BGP decision process [37] and sends BGP messages to its neighbors.

Open*,* Keepalive*, and* Notification *messages are not explicitly simulated, but their effects can be replicated by tearing down sessions.*

Event Queue. *BGPSim* is based on an event queue. In each step, it dequeues the next event, which is then executed on the corresponding router. That router may change its state and push new events back on the queue. Users provide their queue implementation to customize the simulator's behavior. For instance, they can use a deterministic First-In-First-Out (FIFO) queue or use one that models propagation and processing time [116]. This interface is highly flexible and can emulate mechanisms such as the *minimum route advertisement interval* (MRAI) [37] or the batching of BGP routes in a single message.

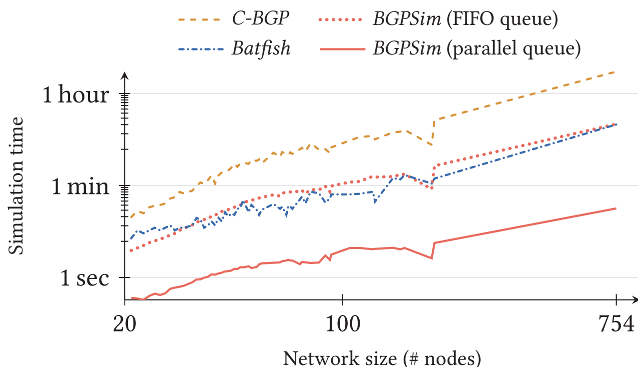


Figure 2.2: BGPSim consistently outperforms C-BGP *and* Batfish *to simulate the network. Even when running on a single core with a FIFO queue, it matches the performance of* Batfish *which relies on all 96 available threads.*

BGPSim also provides a way to speed up the simulation by relying on a parallel queue. Instead of yielding one event at a time, this queue yields all enqueued events for one router. *BGPSim* then executes all these events at once on a single thread, thus minimizing the synchronization overhead.

2.4.2 Evaluation

We evaluate *BGPSim*'s running time to simulate a network and compare it to *C-BGP* [44] and *Batfish* [45]. *BGPSim* consistently outperforms *C-BGP* and *Batfish* in simulating the network by around 460×, and 40×, respectively.

We evaluate the running time for 147 topologies in Topology Zoo [115], ranging from 20 to 754 routers. For each topology, we connect 50 external networks to random routers in the network and generate a configuration based on Gao-Rexford routing policies [38]. After configuring the network, we advertise the same 1 000 prefixes with the same AS path length from all external networks and measure the convergence time.

We run each evaluation on a server with a 64-core AMD Epyc 7747. We let *BGPSim* and *Batfish* use up to 96 threads; *C-BGP* is a single-threaded application. For fairness, we do not measure the time needed to set up the network, as *BGPSim* uses direct API calls, whereas *Batfish* parses configuration files.

The results are shown in Figure 2.2. For the largest topology with 754 routers, *BGPSim* simulates the network in 21 seconds, while *Batfish* takes 15 minutes and *C-BGP* 2.6 hours. When using the regular FIFO queue on a single core, *BGPSim* takes 15 minutes, matching *Batfish*, which utilizes all 96 threads.

In fairness, Batfish supports and simulates more routing processes than BGPSim. Yet, for all these experiments, BGPSim, Batfish, and C-BGP compute the same result.

3 *Guided Exploration of Control-Plane Routing States*

Network verifiers have come a long way in the past two decades, greatly improving in terms of the protocols they support, types of properties they can verify, and scale of the networks they can practically analyze. In contrast to this rapid progress, the actual deployment of control-plane verification tools has been comparatively slower. While multiple reasons explain this slow adoption, we attribute it to the fact that verification systems remain difficult to use [117]. Some related works blame this on verifiers returning *single* counterexamples that *(i)* contain little context about the cause of the problem [46, 118] and *(ii)* often describe unlikely corner cases, even in the presence of a critical misconfiguration [117]. But this only scratches the surface.

Fundamentally, there is a semantic gap between verifiers exploring the space of environments and operators reasoning about routing states. These two viewpoints correspond to the inputs and outputs of the distributed computation of the network: the convergence process finds routing states from environments. This process separates the verifiers' explorations from the user's intent, i.e., the specification expressed as a set of undesired routing behaviors. Thus, verifiers cannot find those counterexamples with the most practical relevance to the users—they would benefit from *guiding* the exploration process themselves to find the relevant, e.g., likely, counterexamples.

Most existing control-plane verifiers rely on generic SMT-based solvers to explore the space of network environments [48, 73, 74]. They generate SMT constraints that relate symbolic variables describing the environments with others describing the routing state. These verifiers leave it up to the solver to explore the joint space of environments and routing states, which often finds unactionable counterexamples [118]. While SMT solvers like Z3 offer sophisticated languages to declare and combine search tactics to influence their exploration [119], designing the right tactics is notoriously challenging [120–122]. It is, thus, infeasible for operators to guide the solver's exploration by designing custom tactics.

Some recently proposed network verifiers simulate how routers propagate and select routes symbolically using Binary Decision Diagrams (BDDs) [49, 64]. In doing so, they operate on the *entire* space of environments to describe *all* routing states at the same time. Thus, they can yield the entire space of counterexamples. However, this space is defined in terms of the environments instead of routing states. This mismatch still leaves users guessing at what routing states to focus on first. Moreover, any attempt at guiding the symbolic execution of these verifiers might yield incorrect results.

In short, the semantic gap between operators reasoning about routing states and network verifiers exploring the space of environments prevents operators from *guiding* their exploration according to their domain knowledge and high-level intent. Yet, doing so significantly improves the usability of these verifiers by enabling them to *(i)* find interesting, e.g., likely, counterexamples and *(ii)* prune parts of the search space that provably do not contain any counterexamples.

To bridge this gap, we propose a technique to directly explore the space of routing states, that is, which routes are selected in the network. Specifically, we introduce the *BGP State Iterator*, a novel backtracking algorithm that explores the space of routing states, guided directly by a specification, domain-specific knowledge, or a probabilistic model. In contrast to existing network verifiers, the *BGP State Iterator* can *(i)* find the entire set of environments that trigger a violation, *(ii)* explore the important, e.g., likely, routing states first, and *(iii)* perform specification-guided exploration.

Backtracking algorithms such as DPLL [123] form the basis of many SAT and SMT solvers like z3 [119].

Turning the problem around by directly exploring the large space of routing states is challenging, though, mainly because the vast majority of states are simply incoherent. From the large space of environments, the network will, in fact, only converge to a tiny subspace of all routing states. As such, the problem becomes synthesizing the set of environments that result in each state, and if this set is empty, then the state is unreachable. To that end, we explore the space of routing states hierarchically. The *BGP State Iterator* defines a tree of *partial* routing states, that is, states in which the selected routes of some routers are yet undefined. We find that only a few selected routes usually suffice to prove that the network cannot converge to selecting those routes. Our backtracking algorithm relies on sufficient conditions for partial routing states to be unreachable.

These partial states, together with the restrictions imposed on the environment, are recursively refined until all routers choose a route.

As we show, the aforementioned conditions are *(i)* effective in pruning the search space, *(ii)* efficient to check using simple set operations, and *(iii)* become both necessary and sufficient once the algorithm reaches a leaf. Our algorithm is thus sound and complete, finding *all* reachable routing states, including a description of the corresponding space of environments.

Contributions. We fully implement the *BGP State Iterator* and evaluate it on realistic configurations. In our evaluation, we demonstrate how network verification greatly benefits from specification-guided exploration. In particular, we find the space of all counterexamples in less than a second where SMT-based verifiers such as [48] would timeout after a day. Beyond this, we outline how, given a model that describes the likelihood of observing specific routing inputs and failures, the *BGP State Iterator* can explore the most likely routing states first. In doing so, we lay the foundations for probabilistic control-plane verification on failures *and* routing inputs, something which was only possible before under node and link failures [23, 68].

We will publically release the source code under a GPLv2 license after an anonymous peer review.

Probabilistic verification is checking if properties to be satisfied with at least probability p

3.1 Overview

The *BGP State Iterator* explores the space of BGP routing states using a backtracking algorithm. It finds all stable routing states of the network and yields them one by one. The iterator also generates the corresponding conditions on the environment, i.e., on routing inputs and failures, for which the network converges to that state. These conditions describe a space of concrete environments and thus form equivalence classes.

We define a *partial* routing state as a partial assignment of selected routes only for a subset of routers. Partial routing states thus describe a set of *concrete* routing states that specify the selected routes of all routers. The *BGP State Iterator* explores the tree of such partial routing states; the root contains no selected routes, while all leaves are concrete states.

Concrete routing states are either *stable*, i.e., there exists an environment that leads the network to converge to that state, or *unstable*. We call a partial state *unstable* if it *only* describes unstable leaves. By detecting unstable states early during the exploration, we massively reduce the size of the tree that we must traverse. To that end, we define a simple but expressive language for describing route maps and other transformations.

This language enables us to devise efficient heuristics to check whether a (partial) routing state is unstable using efficient set operations. These heuristics are necessary but insufficient to discriminate unstable states, but they become sufficient as the iterator reaches a leaf—we only yield stable states.

Finding Stable Routing States. The size of the routing state tree grows exponentially in the number of routers. Yet, only a small subset of routing states are *stable*. During its traversal, our algorithm uses effective heuristics to prune unstable routing states. More precisely, a router selecting a route implies that:

In a network where the n routers may choose amongst r routes, the number of routing states grows $\mathcal{O}(r^n)$.

1. *Propagation*: the neighbor advertises that route,
2. *Advertisement*: the route contains the correct attributes,
3. *Preference*: the route is the most preferred route available.

Consequently, the selection of a route massively restricts the stable choices of other routers. For instance, all routers along a propagation path select the same route. Likewise, any neighbor cannot advertise a more preferred route, preventing that neighbor from selecting it in the first place.

Example. Figure 3.1 depicts an example network consisting of two routers, a and b, connected to two external networks, x and z. The *BGP State Iterator* first constructs the BGP graph to capture the BGP configuration. This BGP graph is a multi-graph in which each edge corresponds to one execution path of a route map. Consider the route map on the session $z \to b$. A route with community 100:1 is accepted with local-preference 200. We capture this as an edge e^1_{zb} labeled with $(\{1\}, \emptyset, 200)$. Here, the first set $\{1\}$ describes all communities that must be present, while the second set describes those that cannot be present (none in this case). A route without community 100:1 but with 100:2 is accepted with a local-preference of 100, resulting in edge e^2_{zb}: $(\{2\}, \{1\}, 100)$. The route map from x to a results in a single edge e_{xa}: $(\emptyset, \emptyset, 100)$. The iBGP session between a and b simply accepts all routes without transformations.

We omit the AS number prefix 100: in the set notation.*

The labels of the iBGP sessions between a and b are $(\emptyset, \emptyset, \emptyset)$, thus matching all routes and leaving the local-preference unchanged.

We then construct the tree of routing states, as shown in Figure 3.2. The first level corresponds to routes that router a can select. We capture those routes in terms of paths in the BGP graph: a can choose, $\langle e_{xa} \rangle$, $\langle e^1_{zb}, e_{ba} \rangle$, $\langle e^2_{zb}, e_{ba} \rangle$, or the empty route $\langle\,\rangle$. As the iterator explores one such path, it constructs conditions for a to select that route in three steps: *Propagation*, *Advertisement*, and *Preference*.

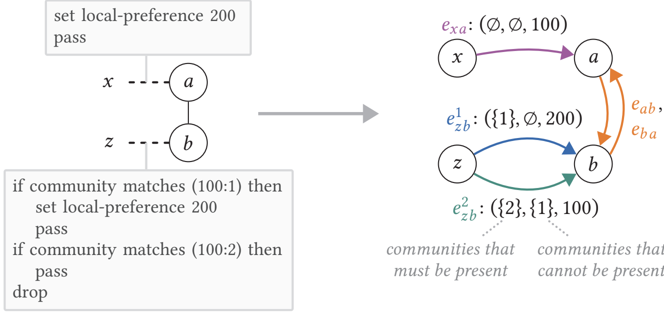


Figure 3.1: The BGP State Iterator transforms a BGP configuration (on the left) to a directed multi-graph (on the right). Each edge represents one possible execution path of the configured route maps.

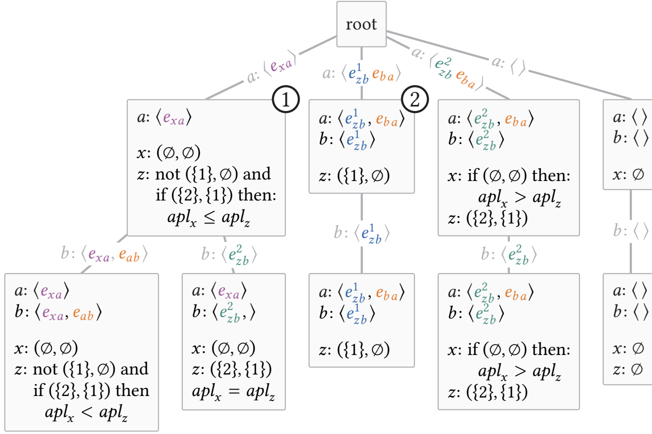


Figure 3.2: We build and explore the tree of (partial) routing states. The first level corresponds to all route choices of router a, while the second one coincides with all remaining options for b. We only show states that are stable, i.e., which we actually explore. Each state contains both selected routes (paths) and conditions on the environment, i.e., on the advertised attributes from x and z.

During *Propagation*, we ensure all routers along the path select the same path. Consider state ②. If a selects the path $\langle e^1_{zb}, e_{ba}\rangle$, then b must also select $\langle e^1_{zb}\rangle$ to propagate $\langle e^1_{zb}, e_{ba}\rangle$ to a. In contrast, for state ①, a selecting $\langle e_{xa}\rangle$ does not yet restrict the selected route of b as it is learned directly from x.

During *Advertisement*, we ensure that the external networks advertise the correct attributes, i.e., communities. Again, let us consider state ①. The path $\langle e_{xa}\rangle$ has no match condition in its route map, and thus, x must advertise *any* route with arbitrary attributes. To select $\langle e^1_{zb}, e_{ba}\rangle$ in ②, z must advertise a route with community 100:1. We write this condition as a tuple $(\{1\}, \emptyset)$; the first set describes all communities that must be present, and the second contains those that must be absent.

During the last *Preference* step, we force any other available route to be less preferred than the selected one. State ② does not restrict advertisements from external network x; the path $\langle e^1_{zb}, e_{ba}\rangle$ is always preferred due to its higher local-preference.

In contrast, for a to select $\langle e_{xa} \rangle$ in state ①, it cannot receive a route from b that is more preferred. To that end, the *BGP State Iterator* generates two conditions. First, it requires that $\langle e^1_{zb}, e_{ba} \rangle$ is not available. Second, if $\langle e^2_{zb}, e_{ba} \rangle$ is available, then the AS path length of x is smaller or equal to the one from z, i.e., $apl_x \leq apl_z$. We allow equality as routes learned from eBGP are preferred over those from iBGP with equal path lengths.

After deciding on a route selected by a, the iterator explores all remaining options for b. From state ①, b can either select $\langle e_{xa}, e_{ab} \rangle$, but only if $apl_x < apl_z$, or $\langle e^2_{zb} \rangle$, which requires that the two AS path lengths are equal. The other options for b result in unstable states. Selecting $\langle e^1_{zb} \rangle$ is prohibited by the conditions inherited from ①, while selecting no route, i.e., $\langle\ \rangle$, fails the Preference condition as b receives $\langle e_{xa}, e_{ab} \rangle$.

Tree Exploration and Use Cases. Our backtracking algorithm allows users to guide the exploration and efficiently solve a wide range of verification problems. In fact, we enable users to specify branches that should be explored first and others to be skipped entirely. This benefits users in the following ways:

We improve the scalability of classical network verification by guiding the exploration according to the specification, that is, only exploring states that could lead to a violation. For instance, to ensure a customer's route is preferred over the one from a provider, we only explore the states in which that customer advertises a route, but some routers still select the route from the provider. In Section 3.5, we demonstrate how the *BGP State Iterator* can find *all* violations of a specification within one second, while *Minesweeper* times out after one day.

By exploring the tree in a depth-first manner, we minimize memory usage and the time required to yield the first state. As an alternative, we can employ more general search techniques to explore the branches that the user is most interested in first. For instance, the operator might want to explore the routing inputs that differ only slightly from the ones currently observed. Moreover, say we are given a probabilistic model that describes the likelihood of observing routing inputs. Using such a model, the *BGP State Iterator* could prioritize the branches with the highest likelihood and find the most likely counterexamples first. This facilitates verifying SLAs like reachability provided for four 9s by visiting the most likely states until the required probability is covered.

Our prototype implementation only supports depth-first traversals. Extending the state iterator to more general explorations is left as future work.

The remainder of this chapter will be structured as follows. We first introduce a model of BGP routes, route maps, and routing states, all of which greatly impact the semantics of the iterator (Section 3.2). We then describe the backtracking algorithm, including our heuristics for determining unstable routing states (Section 3.3). Next, we discuss compatible properties while acknowledging those that the *BGP State Iterator* cannot verify (Section 3.4). Finally, we evaluate the runtime of the *BGP State Iterator* and investigate the impact of guiding the exploration based on a specification (Section 3.5).

3.2 Model of BGP

We model the BGP propagation graph as a directional multi-graph $G_{BGP} = (N, E_{BGP})$ to capture how routes propagate over BGP sessions. This graph describes both BGP sessions and all transformations applied to propagated routes, i.e., route maps and implicit rules from the BGP protocol itself. We use a multi-graph to express each possible execution path of a route map as a distinct edge. We further separate $N = N_i \cup N_x$ into internal routers N_i for which the configuration is known and external networks N_x that may advertise an arbitrary route for the destination d.

The BGP propagation graph is an overlay signaling graph of the physical topology, so BGP sessions might not correspond to links.

In this work, we focus on three important BGP attributes: the *AS path length*, the *local-preference*, and the *community set*. While BGP routes carry additional attributes, these three are representatives of the others. Our model can be extended to support the remaining attributes, either by following similar ideas or by encoding them using special communities. For example, if some routes are denied due to the presence of certain AS in the AS path or because of some IP prefix list, we may encode the presence of such AS numbers or IP prefix ranges through special communities.

Route Maps and Transfer Functions. We introduce a simple and canonical language for expressing route maps and other transformations that are implied by the BGP protocol itself. The language individually captures all possible execution paths of these transformations. We call each such execution path a *transfer function* and use them as edge labels in the BGP graph G_{BGP}.

Let $tf_e = (m_e^\pm, s_e^\pm, lp_e)$ be the transfer function of edge $e \in E_{BGP}$. It encodes match conditions, i.e., $m_e^\pm = (m_e^+, m_e^-)$, and actions, i.e., $s_e^\pm = (s_e^+, s_e^-)$ and lp_e. A statement matches a route if it contains all communities in the set m_e^+ and none of m_e^-. If it does match, the communities in the set s_e^+ are added, while those in s_e^- are removed, and the local-preference lp_i is assigned (or left unchanged if $lp_i = \emptyset$). We require all edges from one router to another to have disjoint transfer functions, i.e., a route can only propagate over one edge from any router to another. Indeed, for any two distinct edges a and b from router u to v:

Notice that we omitted the actions $s_e^\pm$ in the example from Figure 3.1, as the example does not modify the community set.

$$(m_a^+ \cap m_b^-) \cup (m_a^- \cap m_b^+) \neq \emptyset.$$

Consequently, the order of transfer functions is irrelevant. If no statement matches, the route is denied.

Let a path $p = \langle e_0, e_1, \dots, e_{k-1}, e_k \rangle \in \mathcal{P}$ be a propagation path in graph G_{BGP} that propagates a route from external network $src(p) = src(e_0)$ to internal router $dst(p) = dst(e_k)$. We denote $pre(p) = \langle e_0, \dots, e_{k-1} \rangle$ as the prefix of path p without e_k. We call a path *valid* if *(i)* it complies with BGP propagation rules, e.g., routers do not propagate routes learned from an iBGP peer to another one, and *(ii)* there exists a set of route attributes that would match all transfer functions consecutively. We write $tf_p = tf_{e_k} \circ \dots \circ tf_{e_1} \circ tf_{e_0}$ as the composition of all transfer functions along path p. Let $p = \langle a, b \rangle$ be such a path. This path is invalid if the action $s_a^\pm$ of a contradicts with the match $m_b^\mp$ of b, that is, if $s_a^+ \cap m_b^- \neq \emptyset$ or $s_a^- \cap m_b^+ \neq \emptyset$. If the path is valid, the resulting transfer function $tf_p = tf_b \circ tf_a$ of path p is:

$$m_p^+ = m_a^+ \cup (m_b^+ \setminus s_a^+), \quad m_p^- = m_a^- \cup (m_b^- \setminus s_a^-),$$
$$s_p^+ = s_a^+ \setminus s_b^- \cup s_b^+, \quad s_p^- = s_a^- \setminus s_b^+ \cup s_b^-, \quad lp_p = lp_b \,.$$

The two sets $s_b^+ \cap s_b^-$ must be disjoint, making the order of operation for computing $s_b^\pm$ irrelevant.

BGP Routes and Attributes. Any external network $x \in N_x$ may advertise a route for destination d. We model the attributes of these routes using a symbolic representation, describing a set of possible attributes at the same time. We write the attributes that x advertises as $att(x) = (c_x^+, c_x^-, apl_x)$. It is a tuple containing two sets of communities: c^+ are those that must be present, c^- those that must be absent, as well as the AS path length apl_x. Any community outside $c^+ \cup c^-$ is not constrained and may or may not be present without influencing any routing decision. Therefore, $c_x^\pm$ is symbolic and describes a set of community sets. If x does not advertise a route for destination d, we write $att(x) = \emptyset$.

We write $c_x^\pm \vDash m_p^\pm$ if all potential community sets described by $c_x^\pm$ match the conditions $m_p^\mp$ of p's transfer function tf_p. Likewise, we write $c_x^\pm \vDash \neg m_p^\pm$ if no community set described by $c_x^\pm$ matches $m_p^\pm$. Otherwise, some but not all community sets match $m_p^\pm$, which we denote as $c_x^\pm \between m_p^\pm$:

$$c_x^\pm \vDash m_p^\pm \Leftrightarrow \underbrace{c_x^+ \cap m_p^- = c_x^- \cap m_p^+ = \emptyset}_{\textit{no contradiction}} \wedge \underbrace{m_p^+ \subseteq c_x^+ \wedge m_p^- \subseteq c_x^-}_{c_x^\pm \textit{ captures } m_p^\pm}$$

$$c_x^\pm \between m_p^\pm \Leftrightarrow \underbrace{c_x^+ \cap m_p^- = c_x^- \cap m_p^+ = \emptyset}_{\textit{no contradiction}} \wedge \underbrace{m_p^+ \nsubseteq c_x^+ \vee m_p^- \nsubseteq c_x^-}_{c_x^\pm \textit{ does not capture } m_p^\pm}$$

$$c_x^\pm \vDash \neg m_p^\pm \Leftrightarrow \underbrace{c_x^+ \cap m_p^- \neq \emptyset \vee c_x^- \cap m_p^+ \neq \emptyset}_{c_x^\pm \text{ contradicts } m_p^\pm}$$

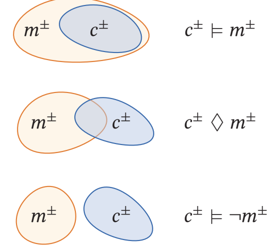


***Figure** 3.3: Illustration of how a symbolic community set $c^\pm$ can relate to a transfer function match $m^\pm$. Both describe sets of community sets. The relations describe how these sets overlap.*

In contrast to external advertisements, we do not represent the attributes of routes selected within the network. Instead, we identify them as propagation paths p in G_{BGP}. We say node $v = dst(p)$ selects p learned from its neighbor $u = dst(pre(p))$ and advertised by the external network $x = src(p)$. In order to reconstruct the v's selected attributes, we apply tf_p on $att(x)$.

3.3 Backtracking Algorithm

The core of the *BGP State Iterator* is a backtracking algorithm. The following explains the construction and traversal of the search tree and our heuristics for pruning it.

3.3.1 Routing State Tree

The *BGP State Iterator* explores a tree of routing states; leaves are concrete states, and all others are partial states. We define a concrete (or partial) routing state $S : N_i \rightharpoonup \mathcal{P}$ as a (partial) function that maps routers to their selected route, that is, a selected propagation path in G_{BGP}. In this chapter, we write $S(v) = p$ and $p \in S$ interchangeably. For any router $v \in N_i$, we define $\mathcal{P}(v)$ as the set of paths in G_{BGP} that start at any external network N_x and end in v. We also add the empty path $\langle\,\rangle \in \mathcal{P}(v)$ to indicate the absence of a route.

We construct a tree of (partial) routing states with a depth of $|N_i| + 1$ as follows: Given an ordering $[v_0, v_1, ...]$ of routers, a partial state at depth k contains a routing state mapping S that is defined for all nodes up to (and including) v_{k-1}. The root at level 0 has an empty state mapping, while leaves at level $|N_i|$ are concrete routing states with fully defined state mappings.

This order affects the algorithm's efficiency. We provide more details in Section 3.3.5.

3.3.2 Stable Routing States

The constructed tree of routing states is enormous, growing exponentially with the size of the BGP propagation graph. Yet, we only need to explore *stable* states—the network only converges to a small subset of possible states. We define three conditions to capture stable routing states: *Propagation*, *Advertisement*, and *Preference*. Propagation requires the neighbor to propagate the selected path. Advertisement ensures the existence of matching routing inputs (attributes). Preference enforces each selected route to be preferred over all available. Formally, for path $p \in S$ to be selected:

$$\textit{Propagation} \quad pre(p) \in S \tag{1}$$

$$\textit{Advertisement} \quad c_x^{\pm} \models m_p^{\pm} \tag{2}$$

$$\textit{Preference} \quad \forall q \neq p \; : \; pre(q) \in S \wedge c_x^{\pm} \models m_q^{\pm} \rightarrow p \prec q \tag{3}$$

$p \prec q$ in Equation (3) states that p to be preferred over any other available path q. The total order relation $\prec$ follows BGP's route-selection algorithm [37]. Our prototype implementation compares the local-preference, AS path length, and IGP cost (breaking ties based on neighboring router ID).

The three conditions in Equations (1)–(3) are necessary and sufficient for any concrete routing state to be stable, i.e., such that there exists a set of attributes *att* in which the network may converge to that state [107].

3.3.3 Heuristics on Partial States

Equations (1)–(3) can be applied to partial states as well by relaxing (ignoring) the conditions on undefined states. The remaining conditions become necessary but not sufficient for a partial routing state to contain at least one stable leaf, i.e., not to be unstable. Partial states that violate the relaxed conditions are proven unstable and can be skipped during exploration, massively reducing the search space.

For each (partial) routing state that the *BGP State Iterator* explores, it performs three steps that correspond to the Propagate, Advertise, and Prefer conditions from Section 3.3.2. In doing so, it accumulates a set of restrictions associated with each partial state and all its descendants. There are three kinds of restrictions: *(i)* AS path length relations, *(ii)* routes that cannot be selected, and *(iii)* routes whose selection implies AS path length relations. Once the iterator reaches a leaf, the accumulated restrictions are equivalent to the conditions from Equations (1)–(3). Therefore, the iterator finds *all stable* routing states, making it both sound and sound and complete.

In the following, we consider exploring the state in which v is assigned to select route $p = \langle e_0, e_1, \dots, e_k \rangle$ from external network $x = src(p)$, advertised by router $u = src(e_k)$.

Propagation. For each sub-path $p_i = \langle e_0, e_1, \dots, e_i \rangle$ of path p with $0 < i \leq k$, we assign $S(n) \leftarrow p_i$ where $n = dst(p_i)$. We then ensure that $S(n)$ is not already set to a different route and that all restrictions collected before are satisfied. The state is unstable and skipped if any restriction for any p_i is violated.

Additional restrictions will be constructed in the subsequent steps.

Advertisement. In the second phase, we ensure there exists a set of attributes $att(x)$ advertised by x, which results in the route p. First, we extend the communities advertised by x to match tf_p: $c_x^+ \leftarrow c_x^+ \cup m_p^+$ and $c_x^- \leftarrow c_x^- \cup m_p^-$. We then check that $c_x^+ \cap c_x^- = \emptyset$. If the two sets overlap, then no such community set exists, and v cannot select p.

Preference. For each path $q = \langle e'_0, \dots, e'_m \rangle \in \mathcal{P}(v) \setminus p$ learned from a different neighbor $n = src(e'_m) \neq u$, we ensure that the route is either not received, or that it has a lower preference, i.e., $p \prec q$. We distinguish two cases. *(i)* If the information in the partial routing state is sufficient to know that v cannot receive q, we can safely ignore route q. Concretely, router v cannot receive q if either $S(n) \neq pre(q)$ or if $c_x^\pm \models \neg m_q^\pm$. *(ii)* Otherwise, we must ensure that $p \prec q$ when v receives q to ensure v selects p over q. Depending on the local-preferences of p and q, we generate different constraints:

- $lp_p > lp_q$ implies v prefers p over q, so we can ignore q.
- $lp_p < lp_q$ implies that v prefers q over p. In that case, we must ensure that v will not receive q, thus adding the restriction: $pre(q) \in S \implies c_x^\pm \models \neg m_q^\pm$.

- $lp_p = lp_q$ implies a relation on the AS path length of the route from x and $y = src(q)$ if v receives q in the first place. More formally, we restrict the $pre(q)$ as follows: $pre(q) \in S \wedge c_y^\pm \models m_q^\pm \implies apl_x < apl_y$. We allow equality if the IGP cost of p is smaller than the one of q (including the tie-breaking rule).

When considering link failures, the *BGP State Iterator* will further fork the state based on the comparison of IGP costs between p and q; see Section 3.3.4. Notice that we do not propagate the selection condition to all sub-paths q of p. These conditions will be checked when exploring $dst(q)$.

Once $pre(q) \in S$ is selected in this or in any subsequent routing state, we add all restrictions to the model associated with $pre(q)$. Such a restriction either denies a certain set of communities to be advertised, i.e., $c_x^\pm \models \neg m_q^\pm$, or it implies that advertising such communities yields a relation of the AS path length. For the latter, we fork the current state into two, one in which $apl_x < apl_y$, i.e., following the AS path relation, and another one in which $apl_x \geq apl_y \wedge c_x^\pm \models \neg m_q^\pm$, i.e., where v does not receive q.

The *BGP State Iterator* stores AS path relations as a directed graph, with edges from one path length to another that must be larger. The presence of a cycle indicates an unstable state unless all relations on the cycle permit equality.

Likewise, the *BGP State Iterator* stores all match restrictions $m_q^\pm$ that advertised attributes cannot match. The currently known advertised communities $c_x^\pm$ relate to such a restriction in three different ways:

- $c_x^\pm \models m_q^\pm$, i.e., the current communities already match $m_q^\mp$. This implies the current partial state is unstable.
- $c_x^\pm \models \neg m_q^\pm$, i.e., the communities already contradict the match $m_q^\mp$. We thus ignore this restriction.
- $c_x^\pm \lozenge m_q^\pm$, i.e., $c_x^\pm$ *might* match $m_q^\pm$. We collect all such restrictions $m_q^\pm$ and use a BDD to check if they "cover" the entire space of $c_x^\pm$, i.e., is there a concrete community set in $c_x^\pm$ that matches none of $m_q^\pm$.

In theory, the *BGP State Iterator* must solve a SAT problem for each explored partial state. Fortunately, most operators do not configure route maps on iBGP sessions, i.e., $tf_q = tf_{pre(q)}$. Thus, $pre(q) \in S$ usually implies $c_x^\pm \models m_q^\pm$, reducing most restrictions to simple and efficient set operations.

3.3.4 Link Failures

Link failures affect the BGP routing state indirectly in two ways. First, failures can partition the network and disconnect BGP neighbors. Second, failures impact the IGP cost and, thus, the route selection process of BGP.

Prior to the exploration, we compute the set of all failure scenarios with up to k link failures. During the exploration, the *BGP State Iterator* updates this set of failure scenarios to contain scenarios that could lead to the current state. When selecting a new route in the Propagation step, we remove all failure scenarios that would disconnect two endpoints of any edge on the path. During the Preference step, when generating the relations on the AS path length, we further check if there exists failure scenarios that make path p have a smaller and larger IGP cost than q. If both exist, we fork the current state into two: one containing all failures in which p has a smaller IGP cost than q, and one in which the opposite is true.

Similar to related works, we only consider the failures that affect IGP costs and paths [23].

3.3.5 Exploration Order

We build the routing state tree by defining an order of routers $[v_0, v_1, \dots]$; Router v_{k-1} is considered when exploring a state at depth k. This order impacts the algorithm's performance. Our heuristics for pruning unstable partial states, i.e., for checking that the partial state only contains unstable leaves, are sufficient but not necessary. The iterator might explore unstable partial states until it eventually proves them unstable. A suboptimal router order can cause more unnecessary explorations.

Our heuristics are not necessary due to the restrictions generated during the Preference step: they only check the selected route of the direct neighbor of a path q, say n, but not all other nodes along q. Extending the restrictions to check the selection on all other nodes is far from trivial, though, as it introduces a long chain of logical implications.

Counterintuitively, we encounter more such situations if we explore routers from the network's edge inwards. Instead, it is beneficial to explore "central" routers, e.g., route reflectors, first. This is because if a node $m \in q$, say another border router, is processed before the direct neighbor n, say a route reflector, then our restrictions on that prohibited path only apply to n and are not yet considered when exploring m.

3.4 Supported Specifications

The flexibility of our algorithm enables us to reason about a wide range of properties and specifications. In the following, we describe properties that the *BGP State Iterator* can verify and outline how it can do so. We also acknowledge properties that our backtracking algorithm *cannot* reason about. Finally, we describe how our algorithm enables probabilistic verification of control-plane properties. We distinguish properties on whether they apply to the routing or the forwarding state.

Routing Properties. These properties relate to the routing state of the network. We follow ideas from Tang et al. [73], who distinguish *safety* and *liveness* properties. Intuitively, safety properties require that "bad" routes can never be selected, while liveness properties require that "good" routes will eventually be selected. Safety properties, such as ensuring that no route from one provider is propagated to another provider, must hold in all states, including *transient* states during BGP's convergence process. The *BGP State Iterator* verifies safety properties by building the BGP graph and enumerating propagation paths. Indeed, if there exists a valid path from one provider to another, there also exists a sequence of BGP messages exchanged that causes such a route leak.

To see this, start with a network that initially knows no routes, start advertising a matching route from the external network, and exchange the BGP messages on this propagation path first.

In contrast, liveness properties pertain to selected routes in the stable state. For instance, ensuring a customer's routes are preferred over those from a provider entails proving that no stable state exists, in which the provider's route is selected while the customer advertises a route for the same prefix. To prove such liveness properties (or to find counterexamples), the *BGP State Iterator* can explore only the partial states in which the border router adjacent to the provider selects the provider's route while the customer advertises one. By further picking a beneficial router order that explores the two border routers first, we only explore a tiny space of the tree and quickly find all counterexamples, as we describe in Section 3.5.3.

Forwarding Properties. Such properties apply to all possible forwarding paths. The *BGP State Iterator* can check properties on forwarding paths by iterating over all stable states and simulating the network in all such states. Our exploration can only be guided by routing properties—forwarding properties often depend on the selected route of many (if not all) routers.

Yet, even for large networks and complex configurations, we show in Section 3.5.2 that the number of stable states remains manageable, especially for recent network simulators [44, 45].

The backtracking algorithm explores *all* stable states. It cannot find routing inputs that cannot converge or reason about *transient* states during convergence. While the former has been extensively studied in the literature [107, 124], the latter remains an open research problem. Some systems measure BGP convergence and how this affects packet loss and other QoS metrics [9, 125–127]. Others perform planned reconfigurations without disturbing traffic during convergence [3, 25, 26, 128].

Like most control-plane verifiers (including [48, 64, 67, 73, 75]), the *BGP State Iterator* is limited to analyzing routes for a single prefix at the same time. The joint space of multiple selected routes simply explodes [23]. Yet, doing so can overlook misconfigurations caused by longest-prefix matching (LPM), where some router along a forwarding path uses a more or less specific prefix. *Expresso* [49] verifies such properties by considering 33 distinct routes simultaneously, one for each IPv4 prefix length. Fundamentally, such problems arise only when a route can be propagated only to a few routers, causing all others to use a less specific route. The *BGP State Iterator* can find all stable states in which only a subset of routers select a route, allowing it to reason about potential issues with LPM.

Likewise, some systems verify link load properties in the network [2, 69, 70]. Doing so, however, requires reasoning about all routes simultaneously—BGP currently distributes routes for over 1 million different prefixes. While the *BGP State Iterator* can reason about path-based forwarding properties such as load balancing, it cannot guarantee that the network remains free of congestion if given an interdomain traffic matrix.

Probabilistic Verification. Probabilistic verification enables the relaxation of specifications, e.g., to maintain guarantees in 95% of scenarios according to SLAs. Our backtracking algorithm opens the door for verifying such properties for both failures and routing inputs. Such specifications involve proving that certain properties are satisfied for at least some probability p. Given a probabilistic model of the environment, the iterator could yield states ordered by their likelihood. We could then explore the tree until we observed either positive examples that cover p or negative ones that cover $1 - p$.

Our prototype implementation only supports depth-first traversals but can easily be extended to support guided exploration of the space of routing states.

3.5 Evaluation

In this section, we evaluate the scalability of the *BGP State Iterator*. First, we analyze how its running time relates to the network size, number of external networks, and configuration—we find *all* stable states of realistic topologies and complex configurations *within hours* at most. Next, we investigate the effectiveness of the specification-guided exploration—we find all counterexamples in *under a second*, exploring less than 0.1% of the space compared to performing a full exploration, while *Minesweeper* [48] times out after 1 day. Finally, we examine how changing the router ordering affects the effectiveness of our heuristics—exploring route reflectors first reduces the number of states explored by up to 10×.

Recall that existing verifiers only aim to find one counterexample.

Implementation. We implement a prototype of the *BGP State Iterator* in ≈7 000 lines of Rust code, including the generation of the BGP propagation graph to represent the configuration and the backtracking algorithm that yields all conditions on the environment. We run all experiments on a server with 96 CPU threads and 256 GB of memory while using only a single thread for each experiment.

Methodology. We use 160 different topologies from Topology Zoo [115], ranging from 20 to 197 routers with up to 243 links. For each topology, we randomly connect routers to external networks and randomly assign them a role: either customer, peer, or provider. We then configure the network to implement the common Gao-Rexford routing policies [38] to make routes from customers preferred over those from peers and providers. By default, we configure the network in an iBGP full-mesh. We use this configuration as a starting point for all experiments while introducing more complex configurations in §3.5.2.

We then measure the time to explore the entire routing state tree to find *all* stable states. Notice that a full exploration shows the worst-case running time of the algorithm, as finding a few counterexamples is usually sufficient. We also count the number of partial states that the iterator explores. We repeat each experiment 100 times and report the median, as well as the 5th and 95th percentile.

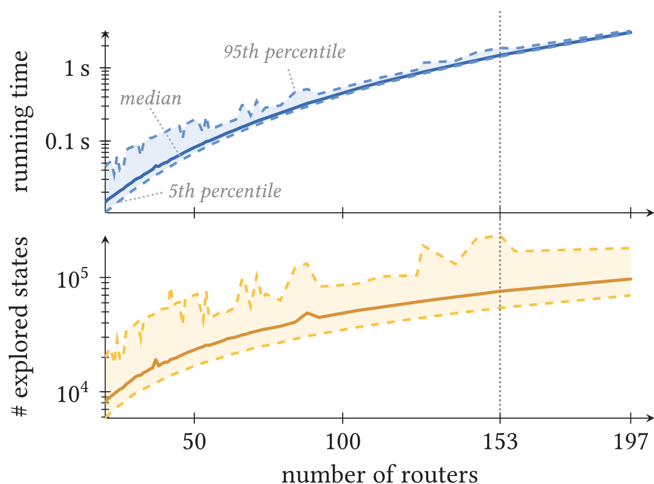


Figure 3.4: *Larger networks make the BGP State Iterator explore more states and, thus, increase its running time. The blue line in the top plot shows the running time, while the orange line in the bottom plot counts the number of explored states. The plot shows the median and the 5th and 95th percentiles.*

3.5.1 Network Size

The network topology influences our algorithm's running time by affecting the size of the routing state tree that it explores. The number of routers in the network increases the depth of the search tree. Adding more external routers increases the number of possible routes and, thus, the number of leaves of each node and the width of the tree. Finally, the number of links impacts the number of failure scenarios to consider and, thus, how expensive each exploration step is. We investigate each dimension separately in the following.

Number of Routers. We first measure how the number of routers affects the algorithm's running time by enumerating all stable routing states of all 160 topologies, each with 10 external networks. We show the results in Figure 3.4 (notice the logarithmic y-axis). Even for large networks of almost 200 routers, the *BGP State Iterator* can find *all* stable routing states in less than five seconds (as shown in the top plot).

The number of states explored is shown in the bottom part of Figure 3.4 and remains below 10^6. To put this number into perspective, let us consider the theoretical maximum: the tree of routing states may have as much as $\sum_{k=0}^{n} r^k$ nodes for n routers and r possible routes. For a network with 100 routers and 10 external networks, the tree already becomes larger than 10^{100}. However, this is a significant overapproximation.

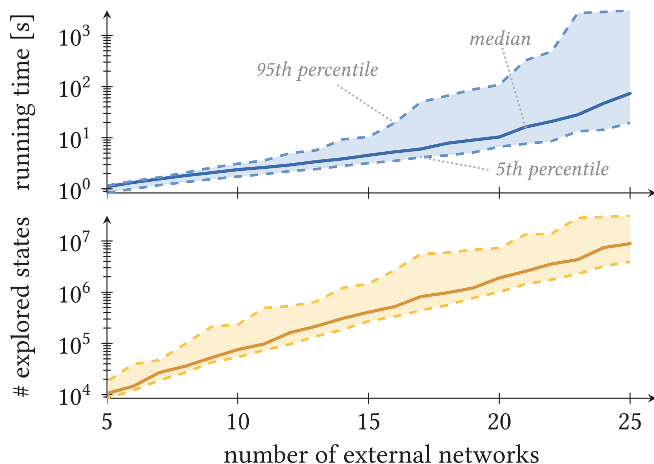


Figure 3.5: *The running time of the BGP State Iterator significantly depends on the number of external networks. The blue line at the top shows the running time, and the orange line at the bottom counts all explored states.*

Once we reach a partial state in which all border routers have selected a route, all others have only one remaining valid choice. An upper bound for the effective tree size is $(n-b)\cdot r^b + \sum_{k=0}^{b} r^k$, in which all b border routers have r possible routes to choose from, while all $n-b$ remaining ones have only one. This yields a tree size of over 10^{12} nodes for networks with 100 routers and 10 external peers, far above the 10^5 states that we explore.

Number of External Networks. We take a single topology, *Colt*, with 153 routers and 177 links and vary the number of external networks from 5 to 25. Figure 3.5 shows the *BGP State Iterator*'s running time (above) and number of states explored (below) using a logarithmic y-axis.

Both the running time and the number of explored states grow exponentially in the number of external networks. Indeed, introducing more eBGP sessions with new neighbors increases the number of possible routes and egress combinations in stable states. Despite that, we usually find *all* stable states within a couple of minutes. In the worst case, it takes the iterator up to 10 hours to explore the entire space.

The large variance in Figure 3.5 can be attributed to the random assignment of external networks to roles (customer, peer, or provider). Routes from networks of the same role have the same local-preference, and hence, a stable state can select routes from any combination of networks of the same role. An even distribution of the roles results in fewer stable states, while a more skewed distribution yields much more.

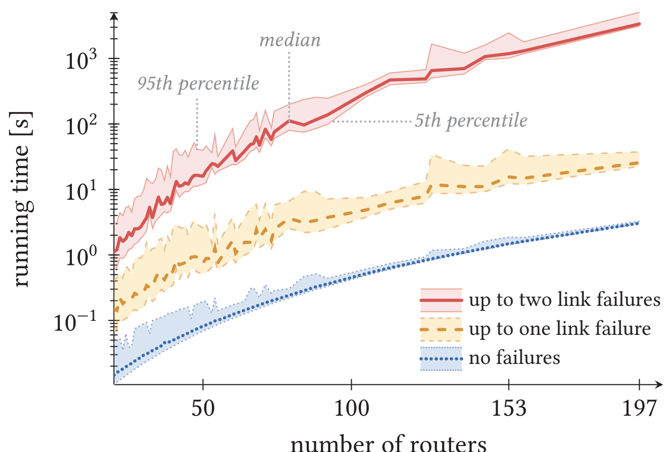


Figure 3.6: *The BGP State Iterator can find all states with up to two link failures within one hour. The green solid line shows the running time when considering all up to two simultaneous link failures. The orange dashed line shows the same for a single failure, and the blue dotted line does so for no failures.*

Link Failures. Exploring the space of both routing inputs *and* failure scenarios has multiple effects on the running time of the *BGP State Iterator*. First, link failures add more stable states, as internal routers can choose different routes depending on the IGP cost. Consequently, it also affects the number of partial states that we explore. Moreover, comparing any two routes, which happens very frequently, becomes more complex: the iterator must consider all remaining failure scenarios.

In Figure 3.6, we measure the time to find all stable routing states of all 160 networks for up to one (dashed orange line) and up to two (solid red line) simultaneous link failures and compare these times with when no failures are considered (dotted blue line). Even for large networks, the *BGP State Iterator* finds all stable routing states with up to two link failures within one hour, including the subset of failures that lead to these states. For the large networks, around 40 k different failure scenarios with up to two link failures are considered, yet the increase in compute time is much less than if we had considered these cases independently.

Takeaways. The *BGP State Iterator* scales to large networks, both in terms of number of routers (we evaluate networks of up to 200 routers) and external networks (we evaluate up to 25 of them), finding *all* states within hours at most. Likewise, our backtracking algorithm can explore up to two simultaneous link failures within hours.

3.5.2 Configuration complexity

In the next set of experiments, we increase the complexity of the configuration by adding common BGP policies. We configure all 153 routers of the *Colt* network and 10 external networks with various policies and find all stable states when considering no link failures. The upper half of Table 3.1 compares the median running time, ae well as the number of states explored and found across five configurations; each builds on the previous one, progressively increasing the configuration's complexity.

We start with the basic *Gao-Rexforxd policies* [38] that set a high local-preference for customer routes, a low one for provider routes, and one in between the two for routes from peers. The policy also tags routes so they can be filtered out on outgoing eBGP sessions. The *BGP State Iterator* finds all 42 states in a few seconds.

Next, we randomly select three routers as *route reflectors*. Every other router is a client of all three reflectors, while the reflectors themselves are connected in a full-mesh. This iBGP topology is common as it maintains reachability even if some reflectors fail [39]. Introducing route reflection does not increase the number of stable states, but it slightly increases the number of explored states. This is because the propagation paths become longer, making our restrictions in the Preference step less effective, c.f., §3.3.5.

On top of this, we consider *No-Advertise*, a well-known community that allows a route to be selected but prevents it from being propagated further [129]. Doing so increases the running time from seconds to minutes and the number of stable states and states explored by two and three orders of magnitude, respectively. Indeed, handling this community *(i)* effectively duplicates all available routes and *(ii)* allows customer and provider routes to be selected at the same time.

Route filters are common in BGP configurations, e.g., to ignore routes with certain ASes along the path. We encode this as a special community—we configure the network to ignore routes advertising that community. Introducing route filters only changes the conditions in the environment but not the number of explored or stable states. The *BGP State Iterator* must thus check more constraints during the exploration, which slightly increases its running time to around 4 minutes.

Configuration	Time	# states explored	# states found
Gao-Rexford policies	2.30 s	75 k	42
+ Route reflection	2.65 s	86 k	42
+ No-Advertise	170.60 s	47 086 k	3 624
+ Route filters	223.45 s	47 086 k	3 624
+ Adjust local-pref	986.10 s	222 476 k	13 247
Verification (correct)	0.67 s	6 k	0
Verification (with bug)	0.88 s	178 k	9
Minesweeper⁻ (correct)		*Timeout after 1 day*	
Minesweeper⁻ (with bug)		*Timeout after 1 day*	

Table 3.1: *Our iterator finds* all *states for complex configurations in under an hour. It verifies such complex configurations within less than a second, exploring less than 0.1% of states compared to the full exploration.* Minesweeper⁻ *fails to solve the same problem within one day.*

Finally, we *adjust the local-preference* of routes from peers if they advertise a special community. Again, doing so increases the number of available routes available and stable states in the network. Yet, we find all 13 k stable states in around 20 minutes by exploring around $2 \cdot 10^8$ partial states.

Takeaways. These experiments demonstrate that the *BGP State Iterator* can find *all* stable states for realistic configurations in a reasonable time.

3.5.3 Specification-Guided Exploration

In all previous experiments, we explored the entire tree of routing states. Yet, for most applications, this is not necessary. Specifically for verifying a network we, only need to traverse the parts of the tree that contain counterexamples. To evaluate the effectiveness of specification-guided exploration, we use the *BGP State Iterator* to verify that valid routes from customers are always preferred over routes from peers and providers.

To that end, we configure the *Colt* network with all five configuration options and policies mentioned in Section 3.5.2. We then traverse only the branches in which a customer's route is not preferred. In essence, we explore the tree multiple times, once for each pair of a customer and a peer/provider. In each iteration, we require the border router adjacent to the peer/provider to select its local route, while the customer must advertise a valid one (without the *No-Advertise* community). Doing so, we will only explore the parts of the tree that can contain violations, see Section 3.4.

We adjust the router order in each iteration to first explore the border router connected to the peer/provider, followed by the one connected to the customer.

The results are shown in the second half of Table 3.1. The *BGP State Iterator* only explores 6 k states in less than a second to prove that the specification is satisfied. Recall that exploring the entire tree necessitates searching through over 10 000× more states. The majority of time is spent building the BGP propagation tree and enumerating all possible routes. The actual exploration takes less than a millisecond.

Next, we introduce a bug in the configuration to evaluate the behavior of our algorithm to find all counterexamples. We do so by adding a new route map to an iBGP session that increases a route's local-preference if a specific community is present. The *BGP State Iterator* still finds all nine counterexamples in less than a second while exploring 178 k states—less than 0.1% compared to exploring the entire search space.

We compare the performance of the *BGP State Iterator* with *Minesweeper* [48], a popular open-source control-plane verifier that is based on an SMT solver. *Minesweeper* supports many protocols and features that are not needed in our experiments. In fairness, we re-implement *Minesweeper*$^{-}$ by expressing only the necessary aspects of the network in SMT constraints. We find that *Minesweeper*$^{-}$ cannot solve the same verification task as it times out after one day.

We only express the BGP routing state in SMT constraints. We do not consider failures and precompute all IGP costs. Further, we do not construct symbolic forwarding paths.

Takeaways. Specification-guided exploration significantly reduces the number of states to explore; we explore less than 0.1% of states compared to full exploration. We find all counterexamples in less than a second, while *Minesweeper*$^{-}$ times out after one day.

3.5.4 Router Orderings

Finally, we compare the impact of different router orderings on the number of explored states and, consequently, on the running time of the *BGP State Iterator*. This order determines which routers are considered first and, thus, the effectiveness of the restrictions generated in the Preference step of our backtracking algorithm. To measure this effect, we compare the number of explored routing states by repeatedly searching for all stable states of the same configuration of the *Colt* network with route reflection while varying the router order. We do so for the different strategies outlined in Section 3.3.5 and compare the resulting distributions.

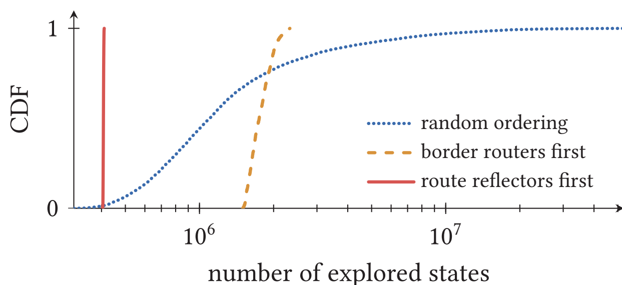


***Figure** 3.7: Choosing a different ordering can reduce the number of explored states by up to 2 orders of magnitude. Exploring the route reflectors first and all border routers last consistently reduces the number of explored states.*

Figure 3.7 demonstrates that carefully crafting the router order can reduce the number of states explored by up to two orders of magnitude. It also confirms our theoretical predictions from Section 3.3.5: Exploring the border routers first causes the algorithm to explore partial states that do not contain any reachable states.

3.6 Conclusion

We explain the lack of usability of stat-of-the-art control-plane verifiers by a *semantic gap* between verifiers exploring the space of environments while users reason about routing states. We propose the *BGP State Iterator*, a versatile algorithm that directly explores the space of routing states. By closing this semantic gap, we enable users to *guide* the exploration, improving its usability in two aspects: First, specification-guided exploration enables us to scale to large networks that are running complex configurations. Second, we open the door for verifying a new class of probabilistic specifications, enabling operators to reason about high-level requirements from SLAs.

We find specification-guided exploration to be extremely effective in pruning the search space. Yet, we also find that the number of possible routing states tends to be relatively small. We attribute this to the fact that most iBGP topologies tend to be shallow (with at most two levels of route reflection), and most iBGP sessions have no (or simple) route maps.

Yet, analyzing performance-related properties requires exploring the routing states of *all* destinations simultaneously, which is simply infeasible. In the next chapter, we explore ways of pruning this space of the joint routing states for all destinations to analyze worst-case link weights.

4 *Verifying Maximum Link Loads in a Changing World*

Link loads – how much traffic crosses each link – are a key indicator of network performance. High link loads, in particular, increase the likelihood of congestion, packet drops, and inflated delays. To meet stringent service-level agreements, operators need to reason about the worst-case load that every link can realistically experience during network operation, e.g., to check that all loads are below a safe threshold and adapt routing configurations if they are not.

Identifying worst-case link loads is hard, though. Operators often have tools to measure traffic and model traffic patterns, even in the long term [130, 131]. However, determining worst-case loads requires scrutinizing how the network forwards traffic in an enormous range of possible events. Indeed, during network operation, internal links and nodes fail. Also, remote failures and policy changes in external networks cause *routing changes*, changing where traffic for remote destinations enters or exits the network. Each of these events causes different link loads: even a single failure or a few specific routing changes can congest currently underloaded links, as our case study on a real ISP network shows (Section 4.6).

Reasoning about link loads is beyond the capabilities of existing network verifiers. Most of them [3, 5, 23, 48, 49, 61, 62, 73] do not support *performance* properties like maximum link loads. They focus on *functional* requirements that pertain to forwarding paths of individual destinations, e.g., the absence of blackholes and forwarding loops. In contrast, performance properties require considering all destinations simultaneously.

A couple of recent contributions [69, 70, 132] focus on assessing properties on link loads. Yet, they only consider network failures and assume fixed external routes, which is fitting for analyzing controlled network environments such as data centers. For most Internet-connected networks, however, *both* failures and routing changes must be considered *jointly* to provide guarantees on link loads.

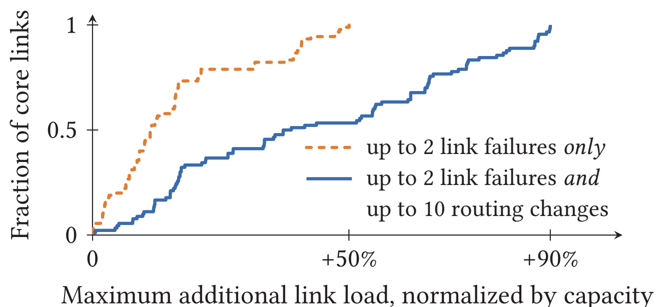


Figure 4.1: *Additional link loads can double due to external routing changes compared with link failures only. This plot shows the additional traffic on all the core links of an ISP network in the worst case when considering only failures (dashed orange) or both failures and routing changes (solid blue).*

As an illustration, Figure 4.1 depicts the additional load on the core links of a real ISP network when simulating up to two simultaneous link failures. In the presence of up to ten routing changes (solid curve), most links reach a higher load than without routing changes (dashed curve), and their load can be twice as big. In other words, ignoring routing changes leads to vastly underestimating maximum link loads.

This paper presents *Velo*, the first verification system that efficiently computes the individual worst-case loads of *all* links for arbitrary link failures and routing changes. *Velo* does not depend on how paths are computed, and hence, it works in both traditional (e.g., IGP/BGP-based) and SDN networks. It also supports typical traffic engineering practices such as Equal Cost Multi-Paths (ECMP) and tunneling (e.g., MPLS).

In each run, *Velo* reports maximum link loads for given router configurations, egress routers, and a traffic matrix. The traffic matrix captures the traffic specification (e.g., a worst-case pattern seen in the past, expected future traffic, etc.) for that run. Different traffic scenarios can be verified in separate runs. *Velo* can limit its computations to user-defined subsets of failures and routing changes that model realistic scenarios. For example, operators can specify a maximum number of simultaneous routing changes and define stable destinations for which routing changes are not allowed.

As a result, *Velo* supports many practical use cases, such as *(i)* verifying configurations before deployment, *(ii)* adaptation of BGP preferences to make the worst-case loads less likely, *(iii)* fine-tuning of network monitors so that they raise alerts only when the risk of congestion is higher, and *(iv)* support for long-term business decisions (e.g., BGP policies). Our case study in Section 4.6 exemplifies some of them.

Challenge. It is infeasible to iterate over all possible routing changes and failure scenarios. For each destination, external networks may advertise new routes, modify existing ones, or completely withdraw them. All these events can change the set of egress routers. In addition, for each destination, the ingress routers and how much traffic enters from them may also change. Such ingress changes depend on updated policies and configurations of external networks. When factoring in possible failures, the search space grows, but it also becomes more challenging to navigate. Indeed, failures generally affect paths to multiple destinations. Hence, destinations cannot be treated as independent from each other across failures. This also prevents us from extending existing verifiers [48, 49, 73].

Solution. We greatly reduce this search space in three steps. First, we provide a compact abstraction of events to consider. We do not explicitly model possible external routes. Instead, we model possible route changes for a destination as sets of *egress routers*. Each set describes one routing state and captures many equivalent routes inducing the same link loads.

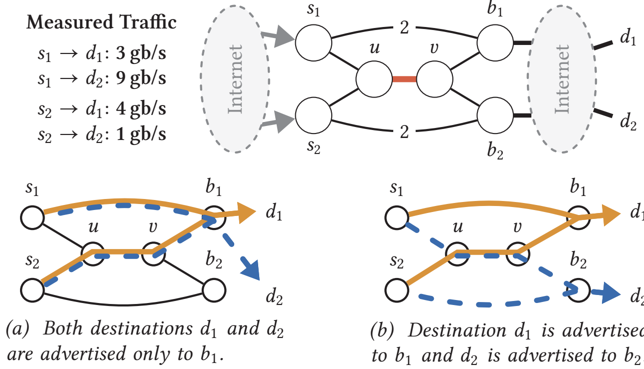
Similar to the ideas presented in Chapter 3, we directly explore the space of routing states. The main difference is that Velo considers a subset of states that provably contain the worst-case state.

Second, we design *Velo* to explore a *minimal* set of states guaranteeing correctness. For common intradomain routing schemes, e.g., shortest-path routing, *Velo* computes all worst-case link loads in *polynomial* time. Such surprising efficiency derives from proving that each link is maximally loaded when, for each destination, only one egress router is used and/or traffic enters from only one ingress router. In the presence of paths deviating from the above schemes (e.g., for traffic engineering), *Velo* extends its search to *minimal* sets of egress routers, ensuring limited impact on *Velo*'s efficiency.

Third, we propose an efficient technique to approximate the input traffic matrix by combining destinations with similar traffic patterns while still maintaining strong guarantees of overall accuracy. We formalize this as a clustering problem, and formally prove that the clustering error gives bounds to the approximation error of the worst-case link load.

Results. Our evaluation on real network topologies shows that *Velo* computes the worst-case load of every link in large networks and all the Internet destinations *within minutes*, even when considering up to two simultaneous link failures. *Velo*'s computed maximum loads are also highly accurate, within 1% of the actual ones, and this across a wide range of traffic matrices.

The source code of Velo is available at github.com/nsg-ethz/velo under a GPLv3 license.



(a) Both destinations d_1 and d_2 are advertised only to b_1.

(b) Destination d_1 is advertised to b_1 and d_2 is advertised to b_2.

Figure 4.2: *Example Network that routes traffic along the shortest paths. Traffic for destinations d_1 and d_2 enters the network at either s_1 or s_2 and leaves at either d_1 or d_2, depending on the received BGP routes. Subfigures (a) and (b) show the forwarding paths for two such advertisements. Link weights are 1, except those from s_i to b_i are 2.*

4.1 *Overview of Velo*

Velo finds the worst-case load for each link by exploring the space of failures and routing changes. Consider the example in Figure 4.2. Routers s_1 and s_2 receive traffic for two remote destinations d_1 and d_2. Depending on the received BGP routes, d_1 and d_2 can be reached via either b_1 alone, b_2 alone, or both simultaneously. Both s_1 and s_2 send packets to the closest egress among b_1 and b_2 that can reach the destination, i.e., implementing hot-potato routing.

Let's initially focus on the link (u, v), assuming no failures and no changes in ingress routers. The link load depends on the BGP routes received by b_1 and b_2. For example, if only b_1 receives BGP routes for d_1 and d_2, packets from s_2 cross the link (u, v) but not those from s_1, resulting in a total load of 5 gb/s. This scenario is depicted in Figure 4.2a.

To find the *worst-case* link load, we conceptually need to check all combinations of each destination being reachable from b_1, b_2, or both. For example, starting from Figure 4.2a, if b_2 receives the same BGP route as b_1 for d_1, then the load on (u, v) decreases as traffic from s_2 to d_1 is moved to the link (s_2, b_2). Figure 4.2b shows the worst-case scenario: when d_1 is only reachable from b_1 and d_2 only from b_2, then 13 gb/s are sent over (u, v) since the two largest demands $s_1 \to d_2$ and $s_2 \to d_1$ both cross that link.

Consider now any other link, for example, (s_2, b_2). Clearly, Figure 4.2b is not the worst-case scenario for (s_2, b_2); its load is higher if s_2 uses b_2 as the next hop for both d_1 and d_2.

Real networks are much bigger than Figure 4.2 and have many more BGP routers and millions of destinations, making the problem challenging to solve. For example, iterating over all the possible BGP routes received by any border router for each destination does not scale. In addition, ingress routers can change as well, which adds another combinatorial dimension. Worse yet, every possible failure affects the network topology, so it is necessary to re-assess the impact of each routing change.

4.1.1 Problem Statement

We now describe *Velo*'s problem and provide the intuition of how *Velo* accurately and efficiently solves it. We refer to any router that receives traffic from outside the network as an *ingress router*, and one that sends traffic outside as an *egress router*. For example, in Figure 4.2b, both s_1 and s_2 are ingress routers, and b_1 is the only egress router for d_1. A *routing change* occurs whenever the ingress or egress routers for a single destination change, such as between Figures 4.2a and b.

> Given *(i)* a network configuration, *(ii)* a traffic matrix, *(iii)* initial egress routers, *(iv)* constraints on routing changes, and *(v)* a space of failure scenarios, find the worst-case load for each link along with the routing changes and the failure scenarios causing such a load.

Network configuration. We define a network configuration as the network topology and the set of all router configurations. We support a wide range of networking paradigms and protocols and present an abstraction in Section 4.2. Operators can specify either the current topology and router configurations as input or alternative ones to perform what-if analyses.

Traffic Matrix. To assess the impact of routing changes, we define a traffic matrix as traffic volumes for each pair of ingress router and destination prefix. In each run, *Velo* takes a single traffic matrix as input. Operators can provide the current traffic matrix or, for more conservative analyses, they can compute such a matrix from one or multiple congestion events recorded in the past. They can also run *Velo* multiple times for different matrices.

We model uncertainty on traffic volumes as QARC does [69]. Namely, we allow operators to specify the total amount of traffic across all the destinations that may need to be forwarded in addition to the input traffic matrix.

Egress Routers. The traffic matrices required by *Velo* include information about per-destination ingress routers but not about egress ones. *Velo* thus requires that operators specify, as a separate input, the set of egress routers for each destination in the absence of routing changes. Once again, operators have the choice to provide current egress routers or hypothetical ones. In the former case, they can simply derive this information by identifying the routers that select as best a BGP route received from an external network.

Routing Changes. In theory, the ingress and egress routers for each destination can all change at the same time. This is not very likely, though. More generally, some routing changes are not worth considering in practice. For example, at operational timescales, receiving new BGP routes for a few destinations is more likely than the same occurring for all destinations. Also, router configurations may only allow stub customers to send BGP routes for prefixes the customers own.

Velo allows limiting the routing changes verified in each run. Operators can indicate the maximum number k of routing changes, where the *number of routing changes* is the number of destinations featuring at least one ingress or egress change. In addition, for each destination, operators can specify whether its ingress or egress routers can change. This distinguishes traffic into four classes:

(i) *transit traffic* where the ingress and egress can change,
(ii) *ingress traffic* towards stub customers with fixed egresses,
(iii) *egress traffic* from stub customers with fixed ingresses,
(iv) *internal traffic* with fixed ingress and egress routers.

Traffic to and from public services in a data center or stub customers is ingress and egress traffic, as their location does not change.

Failure scenarios. In contrast to routing changes, network failures generally affect many forwarding paths for many (if not all) destinations. *Velo* takes as input a description of the space of failure scenarios to explore. This can be the set of up to l simultaneous link or node failures [23, 69] or even include Shared Risk Link Group (SRLG) information [133]. The only constraint here is that for each failure scenario, *Velo* can derive a new topology from the original one.

4.1.2 Velo, in a nutshell

Verifying link loads requires exploring a gigantic search space. At any time, for each destination, any BGP-speaking router can theoretically become a new ingress router or stop being one. Similarly, new BGP routes or changes in their attributes can arbitrarily modify the set of egress routers. On top of this, link failures prevent destinations from being verified independently, i.e., one at a time. A key ingredient of *Velo* is its ability to massively reduce both the search space and the number of destinations. We now provide the intuition of how it does so, referring to the following sections for details.

Reducing the search space. In Section 4.3, we prove that the worst-case link loads occur when, for each destination, all the routers select the same egress router. Indeed, the worst-case load for (u, v) in Figure 4.2 occurs when all routers select b_1's route for d_1 and b_2's route for d_2. This insight enables us to explore only $|P| \cdot |N_B|$ states per failure scenario, where P is the set of destinations, and N_B is the set of egress routers—a polynomial search space instead of an exponential one. The same is true for ingress changes; worst-case loads are achieved when all traffic for one destination enters at the same ingress.

The above property holds for all strictly isotone routing protocols (such as shortest-path routing) that usually govern intradomain routing. We extend *Velo*'s search algorithm to support exceptions to strict isotonicity, such as MPLS tunnels for traffic engineering, by computing the minimal set of additional states to explore in these cases (Section 4.3.1).

Velo iterates over all the input failure scenarios one by one. We experimentally show (Section 4.5) that *Velo* is more efficient than approaches attempting to prune failure scenarios, like QARC, even when considering combinations of up to 3 or 4 simultaneous failures. Intuitively, this happens because *Velo* analyzes any failure scenario *very* quickly, thanks to the search space reduction described above.

Reducing the number of destinations. In Section 4.4, we show how we combine destinations with *similar* traffic distributions across the same ingress routers and obtain an approximated traffic matrix that contains substantially fewer destinations. This approximation significantly improves *Velo*'s running time at the expense of potentially degrading its accuracy.

We present a modified k-means clustering algorithm that computes an approximated traffic matrix with provable bounds on the accuracy loss. In Section 4.5.2, we experimentally show that the approximation is highly accurate for both realistic (heavy-tailed) and unrealistic (almost uniform) traffic matrices. We also show that *Velo*'s clustering technique is more efficient and general than the traditional approach based on heavy-hitters: it indeed achieves tighter error bounds by considering fewer destinations across all the evaluated traffic matrices.

To illustrate our approximation, consider a third destination d_3 in Figure 4.2 with traffic $s_1 \rightarrow d_3$ of 0.4 gb/s and $s_2 \rightarrow d_3$ of 0.3 gb/s. d_3 is also reachable via b_1, b_2, or both. The traffic for d_1 (3 gb/s from s_1 and 4 gb/s from s_2) is ten times larger than for d_3, but their traffic is distributed similarly across ingresses. *Velo* would consider d_1 and d_3 jointly and compute an approximated traffic matrix with the demands for d_1 and d_3 summed up. Doing so would yield a maximum load on link (u, v) of 13.3 gb/s instead of 13.4 Gbps when analyzing d_1 and d_3 individually. In this case, *Velo* accurately bounds the approximation error to 0.1 gb/s.

Dealing with traffic variability. *Velo*'s input can include a traffic matrix *and* an amount y of additional traffic. There are infinite traffic matrices compatible with any given y, as y may be distributed arbitrarily among all entries in the traffic matrix. However, we make two key observations. First, allowing y additional traffic increases the worst-case load of any link by at most y when compared to the input traffic matrix. Second, for any link, its maximum load increases exactly by y whenever that link is crossed by *all* the traffic from any single router r to any single destination d: in this case, increasing the traffic from r to d by y indeed increases the load on that link exactly by y.

Based on these observations, *Velo* accounts for y additional traffic by first computing the maximum link loads for the input traffic matrix and then increasing all the maximum link loads by y. This approach may overestimate the maximum load of a link if no router forwards all its traffic for any single destination over that link in the worst-case scenario for that destination. This condition tends to be rare in practice, i.e., <0.1% of our experiments in Section 4.5, implying that our approach does not affect *Velo*'s accuracy in the vast majority of its runs.

To see why our technique is usually accurate, consider a network with two links of equal weight from u to v. The worst-case load of one link most likely involves a failure of the other one, as this doubles its load. Thus, the worst case tends to involve failures that remove equal-cost paths.

4.2 Model and Notation

We model the network as a graph $G = (N, E)$ with routers N connected by edges E. Border routers $N_B \subset N$ are connected to external networks that advertise BGP routes to destinations P, that is, remote IP prefixes populating the routing tables of the routers. For each destination, the network selects a set of *preferred routes*, corresponding to a set $B \subseteq N_B$ of border routers that will be used as *egress routers*, i.e., to forward traffic towards that destination. This selection can be made by a central controller [134] or in a distributed fashion using iBGP [37]. In the case of iBGP, we assume that *(i)* BGP attributes that affect routes' preference are not modified on any iBGP session, and *(ii)* all routers eventually learn their preferred routes. Several techniques are available to avoid iBGP visibility problems or to prove the absence thereof [48, 73, 135–139].

By focusing on egress routers per destination, we hide most of BGP's low-level details (e.g., its decision process). For any given destination d, we must only consider *(i)* the border routers that *can* receive a route for d, and *(ii)* the possible subsets of border routers that can receive *preferred* routes for d. For the former, we check whether configured eBGP ingress route maps discard all routes for d. For the latter, we analyze the attributes modified by ingress route maps, mainly the Local Preference, that affect the BGP decision process. Other attributes, like the AS path, can take arbitrary values, including those that result in a tie during route selection.

Velo essentially solves a similar problem as the BGP State Iterator: Does there exist routing inputs that make the network converge to the given state, i.e., to select the set of border routers B? Yet, by assuming no iBGP route modifications and visibility problems, Velo must only consider ingress route maps. Notice that we can drop this assumption by relying on the BGP State Iterator to find all possible subsets B.

We denote the input traffic matrix as $\mathbf{M}$. Each element $m_{d,s}$ describes the amount of traffic that enters the network at ingress router $s \in N$ and has destination $d \in P$. We define a column $\vec{\boldsymbol{m}}_d$ of $\mathbf{M}$ as the traffic for destination d, as sourced from all ingress routers N, and we write $|\vec{\boldsymbol{m}}_d|$ to indicate the total traffic towards destination d. The network sends traffic $\vec{\boldsymbol{m}}_d$ for d over the computed paths for egress routers in $B \subseteq N_B$, resulting in load $load_e(\vec{\boldsymbol{m}}_d, B)$ on any link $e \in E$.

Intradomain routing. Forwarding paths are computed from each router to its selected egress router. Many intradomain routing approaches exist. Some rely on a central controller [80, 81], while others use distributed Interior Gateway Protocols (IGPs) [33, 34]. Static routes, source routing, or tunneling can also be used to forward traffic [131].

We model the intradomain path computation using routing algebra [106], as introduced in Section 2.3. Let $(A, \preccurlyeq, \oplus, \bullet)$ be the routing algebra that describes the IGP, with attributes (weights) A, a binary operation $\oplus$, and an order relation $\preccurlyeq$. For example, the commonly used shortest-path-based IGPs [33, 34] are modeled as $(\mathbb{N} \cup \{\infty\}, \leq, +, \infty)$, with attributes corresponding to per-link IGP costs, a binary operation that adds costs, and a binary relation that selects the shorter path.

We write link weights as $w : E \mapsto A$. We denote the optimal s–t path in the routing algebra as $p^*_{s,t} = \langle s, n_1, \ldots, n_i, t \rangle$ with path weight $w_{s,t} = w(s, n_1) \oplus \cdots \oplus w(n_i, t)$. We assume that traffic is split equally across all next hops of each router, i.e., the first hops of its optimal paths. This is consistent with traffic distribution in the presence of ECMP [131].

We say that attributes A are *strictly isotone* if $a \prec b$ implies $c \oplus a \prec c \oplus b$ for any $a, b, c \in W$. This property is both necessary and sufficient to ensure that any sub-path of an optimal path is itself optimal, allowing the paths to be computed using a generalized Dijkstra algorithm (if they exist) [106]. Note that strict isotonicity is common in intradomain routing regardless of the forwarding paradigm, i.e., hop-by-hop vs. source routing vs. Software Defined Networking.

Notice the strict order relation, as compared to the definition of (regular) isotonicity in Section 2.3

Strictly isotone weights offer limited flexibility to achieve traffic engineering [140], so operators sometimes configure *exception paths* (e.g., MPLS RSVP-TE tunnels [141]) that carry packets over manually defined sequences of edges. We denote the set of exception paths as ρ that take precedence over shortest paths. There can be multiple s–t paths in ρ, in which case traffic is equally split among them, similar to ECMP.

4.3 Finding Maximum Link Loads

We achieve scalability thanks to a highly efficient algorithm for finding the worst-case link load in any input topology, allowing us to iterate over a large number of failure scenarios. For any such topology, the worst-case load of a link is the sum of its worst-case loads for each destination considered independently. This is because BGP routes and forwarding paths are specified per destination. To find the worst-case link loads in each topology, we search for the worst-case ingress and egress routers per destination, one destination at a time.

Hereafter, we explain how *Velo* performs such a search for a given topology and a single destination. We first assume the absence of exception paths to present efficient algorithms to find the worst-case ingress and/or egress points. We then extend these algorithms in the presence of exception paths and finally show how to find the worst-case link loads while restricting the number of routing changes.

4.3.1 Per-destination worst-case loads without exception paths

The worst-case load that a single destination imposes on a link in a given topology depends on the kind of traffic—whether the egress or ingress points for that destination can change. Naively exploring the space of possible ingress and egress routers (which massively reduces the space already compared to considering all possible, received BGP routes) requires us to explore $O(2^{2|N|})$ states.

In the following, we prove that only a tiny subset of these states must be explored to find the worst-case link loads: the worst-case load is always achieved when all traffic enters at a *single ingress router* and/or leaves at a different *single egress router.* We first assume the absence of exception paths and focus on ingress traffic, followed by egress traffic, before combining the two results to reason about transit traffic. For each, we present an algorithm to find the worst-case link loads and prove that the algorithm is correct. We then extend our algorithms and proofs in the presence of exception paths.

Worst-case ingress. Let $G = (N, E)$ be a network routing traffic matrix $\vec{\boldsymbol{m}}_d$ for destination d. We focus on the load of link $e \in E$ under a fixed set of egress points B. We aim to find a traffic vector $\vec{\boldsymbol{x}}$ towards d with the same total amount of traffic as $\vec{\boldsymbol{m}}_d$, i.e., $|\vec{x}| = |\vec{\boldsymbol{m}}_d|$ that maximizes the load on link e, i.e., $load_e(\vec{\boldsymbol{x}}, B)$.

Theorem 4.1. *For any traffic $\vec{\boldsymbol{m}}_d$ towards d, there exists a single ingress router $i \in N$ that, if all traffic $|\vec{\boldsymbol{m}}_d|$ enters at i, causes at least the same load on e as with $\vec{\boldsymbol{m}}_d$. Formally:*

$$\forall e \in E \,:\, \exists i \in N \text{ s.t. } load_e(\vec{\boldsymbol{m}}_d, B) \leq load_e(|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i, B),$$

where $\hat{\boldsymbol{u}}_i$ is the unit vector with all zeros except in the element corresponding to the ingress i.

Data: Graph $G = (N, E)$, possible ingress routers N_i, fixed egress routers B, weights $w : E \to \mathbb{R}$, traffic $\vec{\boldsymbol{m}}_d$
Result: Vector of all maximum link loads $\vec{\boldsymbol{\ell}}$ and all maximum fractional link loads $\vec{\boldsymbol{f}}^*$

$\vec{\boldsymbol{f}}^* \leftarrow \vec{\boldsymbol{0}}$

$\vec{\boldsymbol{t}}_s \leftarrow \begin{cases} \hat{\boldsymbol{u}}_s & \forall s \in N_i \\ \vec{\boldsymbol{0}} & \forall s \in N \setminus N_i \end{cases}$ *Initialize traffic at ingress nodes N_i*

$G_{dag} \leftarrow \mathit{ForwardingDAG}(G, w, \mathit{roots} = B)$

for $u \in \mathit{TopoSort}(G_{dag})$ **do**
  $\delta \leftarrow |\mathit{out}(u, G_{dag})|$
  for $e = (u, v) \in \mathit{out}(u, G_{dag})$ **do**
    $\vec{\boldsymbol{t}}_v \leftarrow \vec{\boldsymbol{t}}_v + \vec{\boldsymbol{t}}_u / \delta$
    $f_e^* \leftarrow \max_{s \in N} t_{u,s} / \delta$

$\vec{\boldsymbol{\ell}} \leftarrow |\vec{\boldsymbol{m}}_d| \cdot \vec{\boldsymbol{f}}^*$

Algorithm 4.1: *Find the worst-case ingress routers, including the resulting load for all links in the network at the same time.*

Intuitively, Theorem 4.1 holds as there will be a single ingress router $i \in N$, for which the biggest proportion of traffic will be forwarded along the edge e. Moving all traffic to i will maximize the traffic observed on edge e. In other words,

$$\max_{\vec{\boldsymbol{x}} : |\vec{\boldsymbol{x}}| = |\vec{\boldsymbol{m}}_d|} \mathit{load}_e(\vec{\boldsymbol{x}}, B) = \max_{i \in N} \mathit{load}_e(|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i, B)$$

Proof of Theorem 4.1. Let $f_{e,i} = \mathit{load}_e(\hat{\boldsymbol{u}}_i, B)$ be the fraction of traffic that ingress node i forwards along edge e when routing traffic towards egress routers B. Let $i = \arg\max f_{e,i}$ be the ingress with the largest fraction. For any other ingress $n \neq i$, moving all traffic $m_{n,d}$ to i must maintain or increase the resulting load on e. Doing so for all $n \in N_i \setminus \{i\}$ yields $\mathit{load}_e(\vec{\boldsymbol{m}}_d, B) \leq \mathit{load}_e(|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i, B)$. □

Theorem 4.1 allows us to design an efficient algorithm for finding the worst-case loads on all links for a fixed set of egress nodes B. Algorithm 4.1 first computes the forwarding graph rooted at all egresses in B, yielding a directed, acyclic graph (DAG). We then traverse that DAG in topological order (from the leaves to the roots) to find $t_{u,s}$ as the fraction of traffic originating from ingress s that node u forwards. Algorithm 4.1 computes the maximal fractional load f_e^* on link $e = (u, v)$ by dividing $\vec{\boldsymbol{t}}_u$ by u's degree δ to implement ECMP. Finally, it scales $\vec{\boldsymbol{f}}^*$ by $|\vec{\boldsymbol{m}}_d|$ to find the worst-case load $\vec{\boldsymbol{\ell}}$.

Notice that the algorithm considers only a subset $N_i \subseteq N$ of ingress routers, allowing users to restrict where traffic can enter.

Algorithm 4.1 runs in $O(|N|\log|N| + |N_i||E|)$. By memoizing $\vec{f}^*(E)$ as a function of the egress routers E, *Velo* finds the worst-case link loads for all ingress traffic by running Algorithm 4.1 for each distinct set of selected egress routers.

Worst-case egress. Let $G = (N, E)$ be a network routing traffic matrix $\mathbf{M}$ according to the routing algebra $(A, \preccurlyeq, \oplus, \bullet)$ where A is strictly isotone. We focus on one destination d and the maximal load on link e for the *fixed* traffic $\vec{\boldsymbol{m}}_d$ towards d.

Theorem 4.2. *For any non-empty set of egress routers $B \subseteq N_B$ and any link $e \in E$, there exists a single egress $b \in B$ that, if chosen as the unique egress for destination d, causes at least the same load on link e as the set B. Formally:*

$$\forall e \in E \,:\, \exists b \in B \text{ s.t. } load_e(\vec{\boldsymbol{m}}_d, B) \leq load_e(\vec{\boldsymbol{m}}_d, \{b\})$$

Intuitively, Theorem 4.2 holds by the main property of strict isotonic routing protocols; the sub-path of any optimal path is also optimal. All traffic crossing the edge $e = (u, v)$ must follow a path that is also optimal for v, allowing us to reduce the set of egresses to the one selected by v. In case a router splits its traffic equally among multiple paths, we show that the fraction of traffic it sends across the edge e will not decrease.

Proof of Theorem 4.2. Consider any edge $e = (u, v)$, and let $N_e \subseteq N$ be the set of all nodes that forward *some* traffic along e (blue region in Figure 4.3). Further, let b be an egress router preferred by v, and pick any router $n \in N_e$ that forwards traffic along e.

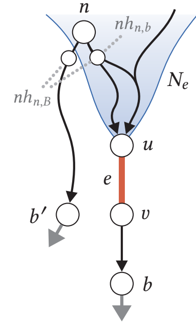


Figure 4.3: *Illustration for the proof of Theorem 4.2. The blue area shows the part of the network that forwards traffic along the edge $e = (u, v)$. There are two egress routers $B = \{b, b'\}$ and node n with two next hops, one in N_e, and one. The arrows show all shortest paths.*

First, observe that the weight of n's optimal path towards any egress router in B, namely $w_{n,B}$, is equal to the weight of its path to b via e, i.e., $w_{n,b} = w_{n,u} \oplus w(e) \oplus w_{v,b}$. Hence, $w_{n,B} = w_{n,b}$ due to the strict isotonicity.

We now show that the total amount of traffic n sends across edge e cannot decrease when only egress b is available. We show this by comparing n's next hops for egresses B, i.e., $nh_{n,B}$, with those for the single egress b, i.e., $nh_{n,b}$. First, we argue that $nh_{n,b} \subseteq nh_{n,B}$, and thus, the traffic n sends to any next hop in $nh_{n,b}$ does not decrease. This follows from the fact that any optimal path of n for b is also optimal for B.

Second, we show that $nh_{n,b} \cap N_e = nh_{n,B} \cap N_e$, proving that the fraction of traffic n sends across the edge e cannot decrease. Consider any next hop $x \in nh_{n,B} \cap N_e$: n can still reach the destination d via egress b on an optimal path via x. This is because $x \in N_e$, and thus, $w_{x,B} = w_{x,b}$. □

Algorithm 4.2: *Find the worst-case egress routers and the resulting load for all links simultaneously.*

```
Data: Graph G = (N, E), border routers N_B ⊆ N,
      link weights w : E → ℝ, traffic m⃗_d
Result: vector of all maximum link loads ℓ⃗
ℓ⃗ ← 0⃗
for b ∈ N_B do
    t⃗ ← m⃗_d                                   initialize traffic on all nodes
    G_dag ← ForwardingDAG(G, w, root = b)
    for u ∈ TopoSort(G_dag) do
        δ ← |out(u, G_dag)|
        for e = (u, v) ∈ out(u, G_dag) do
            t_v ← t_v + t_u/δ
            ℓ_e ← max(ℓ_e, t_u/δ)
```

Theorem 4.2 implies that the worst-case load of any link for any destination is obtained when the entire network chooses a single egress router. More formally,

$$\max_{B \subseteq N_B} load_e(\vec{\boldsymbol{m}}_d, B) = \max_{b \in N_B} load_e(\vec{\boldsymbol{m}}_d, \{b\}).$$

This observation directly enables us to reduce the number of explored states to $O(|N_B|)$ for finding the set of egress routers that maximize the load on any link. Indeed, *Velo* iterates over the border routers in N_B. For each border router b, *Velo* creates the forwarding DAG rooted at b. It then computes the link loads for those border routers by traversing the DAG in topological order and pushing traffic from each node to its outgoing edges.

Algorithm 4.2 shows our pseudo-code. Its computational complexity is $\mathcal{O}(|N_B||N|\log|N|+|N_B||E||P|)$ because the two most expensive operations are (i) computing $|N_B|$ forwarding DAGs with Dijkstra's algorithm [106], and (ii) traversing a DAG in topological order for each destination prefix $d \in P$.

Worst-case ingress and egress. For transit traffic, any routing change can influence both the ingress and the egress. We aim to find an input traffic $\vec{\boldsymbol{x}}$ with $|\vec{\boldsymbol{x}}| = |\vec{\boldsymbol{m}}_d|$ and a set of egress routers B that maximizes $\max_{\vec{x},B} load_e(\vec{\boldsymbol{x}}, B)$.

Data: Graph $G = (N, E)$, ingress and egress routers N_i and N_B, weights $w : E \rightarrow \mathbb{R}$, traffic $\vec{\boldsymbol{m}}_d$
Result: vector of all maximum link loads $\vec{\boldsymbol{\ell}}$ and all maximum fractional link loads $\vec{\boldsymbol{f}}^*$

$\vec{\boldsymbol{f}}^* \leftarrow \vec{\mathbf{0}}$

for $b \in N_B$ **do**
  $\vec{\boldsymbol{f}}^{\boldsymbol{b}} \leftarrow$ *Algorithm 4.1*$(G, N_i, B = b, w)$
  $f_e^* \leftarrow \max(f_e^*, f_e^b)\ \forall e \in E$

$\vec{\boldsymbol{\ell}} \leftarrow |\vec{\boldsymbol{m}}_d| \cdot \vec{\boldsymbol{f}}^*$

***Algorithm** 4.3: Find the worst-case ingress and egress routers and the resulting load for all links.*

Theorem 4.3. *For transit traffic, the load on any link e is maximized when all traffic $|\vec{\boldsymbol{m}}_d|$ enters at a single ingress i and leaves at a single egress d. Formally,*

$$\max_{B \subseteq N_b, |\vec{x}| = |\vec{\boldsymbol{m}}_d|} load_e(\vec{\boldsymbol{x}}, B) = \max_{i \in N_i, b \in N_b} load_e(|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i, \{b\})$$

Proof of Theorem 4.3. By Theorem 4.1, for any set of egresses, the load on link e is maximized if all traffic $|\vec{\boldsymbol{m}}_d|$ enters the network at a single ingress router i, i.e., $|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i$. Further, by Theorem 4.2, for any traffic vector (including $|\vec{\boldsymbol{m}}_d| \cdot \hat{\boldsymbol{u}}_i$), the load on link e is maximized when only a single egress router b is selected. Theorems 4.1 and 4.2 together imply Theorem 4.3. □

Velo uses Algorithm 4.1 to find the maximum fractional load $\vec{\boldsymbol{f}}^{\boldsymbol{b}}$ for a single egress router b. Algorithm 4.3 finds the worst-case load for a given destination d by finding the maximum load for all egress routers $b \in N_B$. We repeat this algorithm for a single destination in $O(|N_B||N| \log|N| + |N_B||N_i||E|)$. Notice that its result $\vec{\boldsymbol{f}}^*$ can be memoized and reused for all other destinations that share the same configuration.

Exception Paths & Traffic Engineering. With exception paths, routers forward some packets along non-optimal paths. Yet, Theorem 4.1 still holds with exception paths. We slightly modify Algorithm 4.1 to first propagate traffic along exception paths. Theorem 4.2 also holds, but only in the special case where all traffic enters the network at a single ingress i, i.e., $\vec{\boldsymbol{m}}_d = k \cdot \hat{\boldsymbol{u}}_i$. This is because BGP is a single path protocol; i will select only one egress, and then either forward traffic along the shortest path or along exception paths. Consequently, Theorem 4.3 also holds in the presence of exception paths.

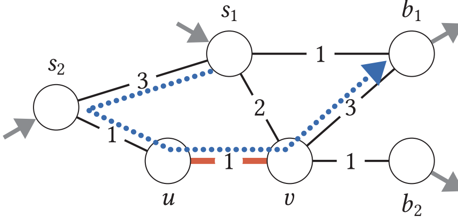


Figure 4.4: Example network with one exception path from s_1 to b_1. Considering fewer combinations of egress routers than Velo leads to underestimating maximum link loads.

Theorem 4.2, however, no longer holds in general. Take the example depicted in Figure 4.4, with two egresses b_1 and b_2, and two ingress routers s_1 and s_2. Packets are forwarded along shortest paths, except s_1 forwards traffic for b_1 on path $\langle s_1, s_2, u, v, b_1\rangle$, drawn as a green dotted line. The load on link (u, v) is maximized when both b_1 and b_2 are egress routers. Indeed, shortest paths are such that s_2 bypasses (u, v) if b_1 is the only egress, and s_1 bypasses (u, v) if b_2 is the only egress.

To handle exception paths for destination d, we explore all single egress routers plus all combinations of the border routers that terminate any exception paths configured for d. The example in Figure 4.4 demonstrates that considering fewer egress routers can lead to incorrect results. We now show that considering only these combinations is sufficient to find the worst-case link loads under egress traffic. Let ρ_d be the exception paths configured for d, and let $T = \{b \mid \langle s, \dots, b\rangle \in \rho_d\}$ be the set of egress routers ending any path in ρ_d.

Theorem 4.4. *For any non-empty set of egress routers $B \subseteq N$ and any link $e \in E$, there exists a subset $B' = \{b\} \cup T'$, where $T' = T \cap B$, that causes at least the same load on e than the set B. Formally, for every link $e \in E$,*

$$\exists b \in B \text{ s.t. } load_e(\mathbf{M}_d, B) \leq load_e(\mathbf{M}_d, \{b\} \cup T \cap B)$$

Intuitively, Theorem 4.4 holds as nodes either forward via optimal paths, in which case Theorem 4.2 applies, or via exceptional paths that are still available in B'.

Proof of Theorem 4.4. Consider edge $e = (u, v)$, and let $b \in B$ be an egress router preferred by v. Now, take any node $n \in N_e$ that forwards traffic along the edge e for destination d.

If n forwards traffic along optimal paths (i.e., not using exception paths), then n will not decrease the traffic it sends along edge e by Theorem 4.2. Otherwise, n forwards traffic

Algorithm 4.4: *Find the worst-case link load on all links for optimal or exception paths.*

Data: Graph $G = (N, E)$, border routers $N_B \subseteq N$, link weights $w : E \mapsto \mathbb{R}$, traffic $\mathbf{M}_d$, paths ρ_d
Result: vector of all maximum link loads $\vec{\ell}$

$T \leftarrow \{b \mid (s, \cdots, b) \in \rho_d\}$
$\vec{\ell} \leftarrow \vec{0}$
$y[u,v] \leftarrow 0 \; \forall (u,v) \in E$
$G^b_{dag} \leftarrow ForwardingDAG(G, w, root = b) \; \forall b \in N_B$

for $B \in selected_combinations(N_B, T)$ **do**
    $G_{fw} \leftarrow merge(G^b_{dag} \; \forall b \in B)$
    $\vec{t} \leftarrow \vec{m}_d$
    $\vec{\ell}' \leftarrow \vec{0}$ *Traffic imposed by exception paths*
    for $path = (s, n_1, \cdots, n_i, b) \in \rho_d$ **do**
        if *s forwards to b in* G_{fw} **then**
            $\ell'_e \leftarrow \ell'_e + t_s \; \forall e \in path$
            $t_s \leftarrow 0$
    for $u \in TopoSort(G_{dag})$ **do**
        $\delta \leftarrow |out(u, G_{dag})|$
        for $e = (u,v) \in out(u, G_{dag})$ **do**
            $t_v \leftarrow t_v + t_u/\delta$
            $\ell_e \leftarrow \max(\ell_e, \ell'_e + t_u/\delta)$

along exception paths in ρ_d. In that case, n will use the same forwarding paths for egresses $\{b\} \cup T \cap B$ as all destinations of n's forwarding paths are within T and B. Thus, n will still forward the same amount of traffic over link e. □

Theorem 4.4 allows us to find the worst-case link loads by iterating over all subsets $T' \subseteq T$ and one other egress router in $N_B \setminus T$; see Algorithm 4.4. The function *selected_combinations* filters out combinations of egresses that cannot receive equally preferred routes due to BGP policies, i.e., it yields combinations of border routers that *might* set the same local-preference value.

For each set B of egress routers computed as above, we construct the corresponding forwarding DAG by combining the forwarding DAGs rooted at all egresses in B. We then find the load resulting from traffic routed along exception paths and push the remaining traffic through the DAG as in Algorithm 4.2. Computing link loads in the presence of exception paths has complexity $O(|N|^2 \log|N| + 2^{|T|}|N||E||P|)$.

4.3.2 Restricting the Number of Routing Changes

So far, we have discussed finding the worst-case link loads considering any set of routing changes, that is, ingress and/or egress changes. Part of *Velo*'s input is the maximum number k of routing changes. A single routing change can affect the ingresses, the egresses, or both at the same time.

For each link, we must find the k destinations that cause the highest increase of its load when changing their ingress and/or egress. To that end, we maintain, for each link, a heap of size k with the largest difference between the worst-case and current-state link loads. Maintaining the heap has a time complexity of $\mathcal{O}(|E| \cdot |P| \cdot \log k)$, which is negligible with respect to the total time of the algorithm.

We also store the worst-case routing changes in that heap to reconstruct the worst-case state for each link.

4.4 Approximating the Traffic Matrix

The algorithms presented in Section 4.3 scale linearly in the number of destinations. It is not rare that routers have over one million entries [131, 142], so finding the worst-case link load for this many destinations would be a limiting factor.

We greatly reduce the number of destinations by combining those with similar traffic patterns. Specifically, we approximate the input traffic matrix with a smaller one, leveraging the low effective rank that traffic matrices typically exhibit [143, 144].

We formulate this traffic matrix approximation problem as a clustering problem and find that the clustering error bounds the *approximation error* δ; by how much the link loads differ when computed on the original and approximated traffic matrices. We envision that our approximation technique can be useful within other networking problems for which the size of traffic matrices limits scalability.

Approximation problem. Given a traffic matrix $\mathbf{M} : |N| \times |P|$ and a target of clusters $|C|$, we aim to find a clustering C of the destinations to reduce $\mathbf{M}$ while bounding the resulting approximation error. More formally, we aim to compute a partition C of P, an approximate traffic matrix $\mathbf{A}$ of size $|N| \times |C|$, and an *error bound* ε for the approximation error δ, such that

We take the number of clusters $|C|$ as input. Increasing $|C|$ reduces both ε and δ at the cost of running time. Operators can tune $|C|$ to balance accuracy and scalability.

$$\delta = \max_{e \in E} |max\,load_e(\mathbf{M}) - max\,load_e(\mathbf{A})| \leq \varepsilon, \qquad (1)$$

where $max\,load_e(\mathbf{M})$ is the maximum load of link e for $\mathbf{M}$.

We must, however, be careful to only cluster destinations that the network treats the same. For example, we cannot aggregate destinations that have different exceptional paths or compute optimal paths according to different link weights. To this end, we provide a custom definition of an *Equivalence Class* and only cluster destinations in the same equivalence class. We distribute the available clusters $|C|$ across all equivalence classes proportional to the amount of traffic they carry. Specifically, any pair of destinations in an equivalence class must satisfy the following three conditions:

1. The same egress routers are used for both destinations in the current routing state.
2. The forwarding paths for both destinations are always equal for the same set of preferred egress routers.
3. The traffic for the two destinations must be the same kind, i.e., ingress, egress, transit, or internal traffic.

In the following, we detail our clustering algorithm for destinations in the same equivalence class.

Normalized clustering. We cluster together destinations with similar traffic distribution across ingress routers, i.e., if similar fractions of traffic for two destinations enter from the same routers. Such destinations *likely* produce worst-case link loads for the same routing inputs.

We define the traffic distribution $\vec{\boldsymbol{m}}_d/|\vec{\boldsymbol{m}}_d|$ of destination d as the fraction of traffic from each ingress compared to d's total traffic load $|\vec{\boldsymbol{m}}_d|$. We then use the L1 norm [145] to measure the similarity of two distributions, as the L1 norm quantifies their absolute difference. We run an *adaptation* of the k-means clustering algorithm on the traffic distributions: instead of minimizing the L1 distances within a cluster c_i, we weight the distance between its cluster centroid $\vec{\boldsymbol{a}}_i$ and a destination d according to d's total traffic. Thus, the computed clusters accurately resemble high-traffic destinations. Cluster centroids $\vec{\boldsymbol{a}}_i$ and the clustering error ε are defined as follows.

$$\vec{\boldsymbol{a}}_i = \sum_{d \in c_i} \vec{\boldsymbol{m}}_d \qquad \varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} |\vec{\boldsymbol{m}}_d| \cdot \left| \frac{\vec{\boldsymbol{m}}_d}{|\vec{\boldsymbol{m}}_d|} - \frac{\vec{\boldsymbol{a}}_i}{|\vec{\boldsymbol{a}}_i|} \right|. \tag{2}$$

Finally, *Velo* constructs the approximate traffic matrix by simply concatenating all cluster centroids, i.e., $\mathbf{A} = [\vec{\boldsymbol{a}}_1, \vec{\boldsymbol{a}}_2, \cdots]$.

Our clustering approach guarantees that the approximation error δ is bounded by the clustering error ε. Intuitively, this is because *(i)* ε measures the traffic in $\mathbf{M}$ that the approximation $\mathbf{A}$ does not account for, and *(ii)* ε is also the additional traffic in $\mathbf{A}$ that is not part of the original matrix $\mathbf{M}$. Consequently, $max\,load_e(\mathbf{A})$ can differ from $max\,load_e(\mathbf{M})$ by at most ε.
In the following, we formally prove that the clustering error bounds the approximation error of the worst-case link load. To that end, we first introduce two lemmas that pertain to linearity and the superposition of maximum link loads before proofing Theorem 4.5.

Lemma 4.1. *$max\,load_e(\mathbf{M})$ is linear in terms of $\mathbf{M}$, i.e.,*

$$\forall a \geq 0 \,:\, a \cdot max\,load_e(\mathbf{M}) = max\,load_e(a \cdot \mathbf{M})$$

Proof. Lemma 4.1 holds because scaling the traffic matrix does not affect the worst-case routing state, and thus, the forwarding paths in the worst case remain unchanged. As a result, The worst-case load is also scaled by the same factor a. □

Lemma 4.2. *$max\,load_e(\mathbf{M})$ is equal to the superposition of the maximum link loads for all columns of $\mathbf{M}$ under the same failure scenario. In other words,*

$$max\,load_e(\mathbf{M}) = \sum_d max\,load_e(\vec{\boldsymbol{m}}_d)$$

Proof. Lemma 4.2 is true because the forwarding decision for one destination is independent of all other destinations in a fixed topology. We can thus find the worst-case contribution of each destination on link e independently of the others. The sum of all worst-case contributions is thus equal to the worst-case link load on e. □

Theorem 4.5. *The approximation error δ, i.e., the difference in maximum link loads of the original traffic matrix $\mathbf{M}$ and the approximation $\mathbf{A}$, is bounded by the clustering error ε. Formally,*

$$\delta = \max_{e \in E} |max\,load_e(\mathbf{M}) - max\,load_e(\mathbf{A})| \leq \varepsilon. \tag{3}$$

Proof. We prove Theorem 4.5 by considering each individual destination d and its contribution to both the clustering and the approximation error:

We define the approximation of d's traffic $\overline{\boldsymbol{m}}_d = |\vec{\boldsymbol{m}}_d| \cdot \vec{\boldsymbol{a}}_i / |\vec{\boldsymbol{a}}_i|$ in **A** in terms of the centroid of d's cluster i. We can express the clustering error ε as defined in Eq. (2) in terms of $\overline{\boldsymbol{m}}_d$:

$$\varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} \left| \vec{\boldsymbol{m}}_d - |\vec{\boldsymbol{m}}_d| \frac{\vec{\boldsymbol{a}}_i}{|\vec{\boldsymbol{a}}_i|} \right| = \frac{1}{2} \sum_{d \in P} |\vec{\boldsymbol{m}}_d - \overline{\boldsymbol{m}}_d| .$$

The maximum link load $max\,load_e(\mathbf{A})$ can also expressed in terms of $\overline{\mathbf{M}}$ by relying on the definition of $\vec{\boldsymbol{a}}_i$ in Eq. (2) and the linearity and superposition of maximum link loads. We can apply Lemma 4.2 as we keep the same failure scenario.

$$\begin{aligned} max\,load_e(\mathbf{A}) &= \textstyle\sum_{c_i \in C} max\,load_e(\vec{\boldsymbol{a}}_i) \\ &= \textstyle\sum_{c_i \in C} \sum_{d \in c_i} max\,load_e(\vec{\boldsymbol{a}}_i) \cdot |\vec{\boldsymbol{m}}_d| \,/\, |\vec{\boldsymbol{a}}_i| \\ &= \textstyle\sum_{d \in P} max\,load_e(\overline{\boldsymbol{m}}_d) = max\,load_e(\overline{\mathbf{M}}) \end{aligned}$$

This equation holds for any fixed failure scenario. Thus, the worst-case failure scenario for $\mathbf{A}$ *and* $\overline{\mathbf{M}}$ *is identical.*

Now, let's focus on the maximum link load for $\vec{\boldsymbol{m}}_d$ and $\overline{\boldsymbol{m}}_d$ on link e. To that end, we consider traffic x_d^+ that is part of the approximation $\overline{\boldsymbol{m}}_d$ but not in the original $\vec{\boldsymbol{m}}_d$, and traffic x_d^- of $\vec{\boldsymbol{m}}_d$ that is not part in $\overline{\boldsymbol{m}}_d$:

$$\begin{aligned} x_d^+ &= \textstyle\sum_s \max(0, \overline{m}_{d,s} - m_{d,s}) \\ x_d^- &= \textstyle\sum_s \max(0, m_{d,s} - \overline{m}_{d,s}) \end{aligned}$$

Let's first focus on x_d^+, but the argument for x_d^- is symmetrical by swapping $\mathbf{M}$ and $\overline{\mathbf{M}}$. The difference of maximum link loads for $\overline{\boldsymbol{m}}_d$ and $\vec{\boldsymbol{m}}_d$ is, at most, the additional traffic x_d^+ of $\overline{\boldsymbol{m}}_d$, i.e., $max\,load_e(\overline{\boldsymbol{m}}_d) - max\,load_e(\vec{\boldsymbol{m}}_d) \leq x_d^+$. That is because $max\,load_e(\mathbf{M}_d)$, by definition, is the maximum load achievable on link e, and adding traffic x_d^+ cannot increase that maximum by more than x_d^+. Therefore, $\delta \leq x_d^+$ and $\delta \leq x_d^-$.

This argument holds not only for the routing inputs but also for failure scenarios. Assume that the worst-case failure scenario for $\mathbf{M}$ is different from the one in $\mathbf{A}$. The difference in maximum link loads is still at most $x_d^\pm \leq \delta$.

Finally, we show that $\sum_d x_d^+ = \sum_d x_d^- = \varepsilon$. By definition of the L1 norm, $x_d^+ + x_d^- = |\overline{\boldsymbol{m}}_d - \vec{\boldsymbol{m}}_d|$, and that the centroid $\vec{\boldsymbol{a}}_i = \sum_{d \in c_i} \overline{\boldsymbol{m}}_d$ contains equal positive and negative errors, i.e., $x_d^+ = x_d^-$. This proves Theorem 4.5. □

4.5 Evaluation

We evaluate *Velo*'s running time and accuracy. In Section 4.5.1, we discuss the factors that influence *Velo*'s running time. We show that *Velo* can find the maximum loads for all the links in large networks (with almost 2000 links) within a few hours when considering up to two failures and ten routing changes. We also compare *Velo* against *QARC* [69]: *Velo* is much faster than *QARC* for practical scenarios, including up to two failures. In Section 4.5.2, we evaluate the impact of the traffic matrix on *Velo*'s accuracy. We show that *Velo*'s approximation error is less than 1%, even for unrealistic traffic matrices.

Notice that QARC can only verify scenarios with internal traffic, but not with ingress, egress, or transit traffic, as Velo can.

Implementation. We implement *Velo* in $\approx$ 8 000 lines of Rust, including optimized versions of Algorithms 4.1–4.4 and our normalized clustering algorithm. We run the evaluation on a server with 96 CPU threads and 384 GB of memory.

Networks. As a dataset, we take the 75 largest networks from TopologyZoo [115], ranging from 40 to 754 nodes with 80 to 1 790 links. For each network, we randomly connect external peers that can advertise a BGP route to any destination. We then assign random IGP weights to each link.

Traffic matrices. We obtain measured traffic matrices from the research network *Switch*. We also generate synthetic traffic matrices following the gravity model [144, 146] to evaluate *Velo*'s sensitivity to controlled variations of traffic patterns.

4.5.1 Running Time

Several factors affect *Velo*'s running time: network size, number of simultaneous failures l, the maximum number of routing changes k, number of clusters $|C|$, number of border routers $|N_B|$, and number of exception paths $|\rho|$. The specification of traffic additional to the traffic matrix instead has no noticeable impact, consistently with *Velo*'s inexpensive approach, cf. Section 4.1.

We find that the network size and the number of failures have the biggest impact on *Velo*'s running time. We, therefore, divide our experiments into two sets: a first set considering only these two factors and another set considering all the others. We then compare *Velo* against *QARC* [69], a state-of-the-art system that finds link load violations caused by failures.

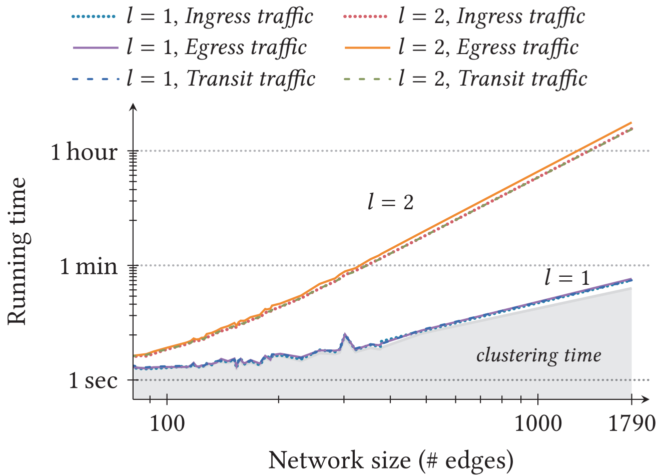


***Figure** 4.5: Velo can scale to large networks of over 1 000 links while exploring up to two simultaneous link failures. This log-log plot shows the maximum running time of Velo for the 75 largest networks in TopologyZoo with up to one (l = 1) and two (l = 2) link failures. The gray area below the curves shows the portion of Velo's running time spent on clustering the traffic matrix.*

Network size and link failures. In the first set of experiments, we analyze the impact of network size and maximum number of failures on *Velo*'s efficiency across all the topologies in our dataset. We evaluate the same scenarios using only ingress, egress, and transit traffic. We set a relatively low number of simultaneous failures ($l \leq 2$) and routing changes ($k = 10$), as it is unlikely for many more links to fail and many more external routes to change at the same time [23]. We also set $|C|$ to 300, as it empirically provides a good accuracy-performance tradeoff (see Section 4.5.2). In these experiments, we configure 30 border routers and no exception path.

Figure 4.5 shows a log-log plot of *Velo*'s overall running time. Varying the network size from 100 to 1 790 increases the running time by about three orders of magnitude. A similar effect is caused by considering all the combinations of two link failures ($l = 2$) rather than all the single-link ones ($l = 1$).

Yet, our results show *Velo*'s practicality in real networks. In the largest TopologyZoo network, *Velo* finds the worst-case load for all its 1 790 links within one minute for single link failures and within three hours for two link failures. In all the other networks, *Velo* finds all the worst-case loads for $l \leq 2$ within two minutes. The asymptotic growth of the running time is roughly cubic for $l = 2$. These findings hold across all the traffic classes: analyzing egress traffic is only 23% slower on average than ingress and transit traffic.

Dimension	Change	Running time change for: Ingress	Egress	Transit
#routing changes	10 → 100	+14.8%	+22.1%	+ 9.7%
#routing changes	10 → ∞	−93.2%	−80.6%	−93.8%
#clusters	300 → 1000	+ 9.4%	+47.2%	+ 9.9%
#in / egresses	30 → 100	+ 4.3%	+47.4%	+ 9.6%
#initial states	30 → 100	+ 4.8%	+ 1.5%	+ 1.6%
#exception paths	0 → 100	− 0.4%	+14.9%	− 0.4%

***Table** 4.1: Velo's running time also depends on the number of routing changes, clusters, ingress and egress routers, initial states, and exception paths. Their effects are negligible compared to the network size and the number of link failures.*

Breaking down *Velo*'s performance, clustering the traffic matrix for the largest network takes under 30 seconds, drawn as the gray area in Figure 4.5. When considering single link failures, *Velo* spends the majority of the time clustering the traffic matrix, performing the actual analysis in only 10 seconds. For two link failures, however, the clustering time is negligible.

A key role in *Velo*'s time efficiency is played by the size of its search space. An estimation follows of the performance improvement provided by such a space reduction. With 30 border routers, a naive approach would require exploring 2^{30} states—seven orders of magnitudes bigger than *Velo*'s search space. Hence, analyzing a network of only 80 links without *Velo*'s state reduction would take roughly one year for $l = 2$.

Other factors. To measure the impact of the other factors, we focus on *Cogent*, one of the largest networks in Topology Zoo, and vary each factor, one at a time. Table 4.1 shows the difference in *Velo*'s median running time for analyzing ingress, egress, and transit traffic with up to $l = 2$ link failures.

The number of route changes k has a small effect on *Velo*'s efficiency, as we always compute the worst-case routing input for *all* destinations before picking the k destinations causing the highest load. Increasing k from 10 to 100 increases the running by up to 30% for maintaining larger heaps. In contrast, allowing all routing inputs to change, i.e., $k = |P|$, *reduces* the running time by up to 90%, as *Velo* needs to maintain less state.

The number of clusters $|C|$ influences *Velo*'s running time primarily for egress traffic, increasing the running time by 50% when using 1000 clusters instead of 300. The runtime increase is much smaller for ingress and transit traffic since the worst-case ingress routers do not depend on the traffic distribution, and hence, *Velo* memoizes the worst-case results between clusters. Deactivating clustering entirely increases the running time by 70× for egress traffic and 14× for ingress and transit traffic.

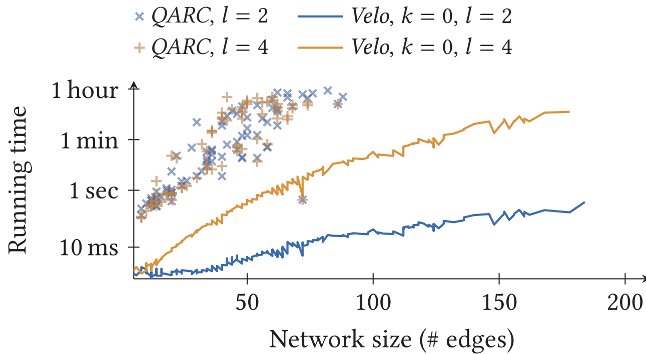


Figure 4.6: *Running time of Velo and QARC [69] for finding link load violations in TopologyZoo networks, without BGP route changes. Velo outperforms QARC for both $l = 2$ and $l = 4$, although the gap becomes smaller.*

The number of ingress and egress routers also impacts *Velo*'s performance, as it affects the number of states to be explored. The effect is more significant for egress traffic, increasing the running time by 50% as it iterates over more egresses. Increasing the number of initial routing states marginally increases the cunning time for ingress traffic only.

Algorithm 4.1 runs once for each set of egresses.

Adding exception paths forces *Velo* to explore additional states, but only for egress traffic (see Theorem 4.2). Since these paths are usually configured manually and used for traffic-heavy destinations, we experiment with up to 100 exception paths. For egress traffic, *Velo*'s running time increases by 15%.

We compute exception paths by randomly permuting link weights.

Comparison with QARC. *QARC* only supports verification of internal traffic, i.e., without changes in routing inputs. We thus compare *Velo* against *QARC* when both systems analyze link loads for internal traffic under l link failures. We re-run the same experiments as in [69] with the author's implementation of *QARC*. These experiments involve traffic between all pairs of routers and 10% additional traffic. We set $k = 0$ in *Velo*, which disables our clustering algorithm.

The results for two and four simultaneous link failures are shown in Figure 4.6. *Velo* is several orders of magnitude faster than *QARC*, achieving sub-second computation across all experiments with $l = 2$. We believe that such a performance gap is mainly due to the different approaches of the two systems: *QARC* solves an Integer Linear Program, while *Velo* relies on Dijkstra's algorithm. *Velo* is 10-100× faster than *QARC* even for $l = 4$, but the performance gap between the two systems seems to progressively decrease with l. We expect that *QARC* will outperform *Velo* for large values of l.

In fairness, QARC guarantees the exact computation of link loads, while Velo theoretically doesn't. However, without clustering, Velo is fully accurate, except for possibly overapproximating additional traffic (see Section 4.1.2). After double-checking, we find that Velo has no approximation error in any of these experiments.

Takeaways. *Velo* finds the worst-case loads of all links within minutes for the most likely and practical scenarios with up to two link failures and a limited number of route changes. Because it is mostly polynomial in the network size, *Velo* scales to large networks with close to 2 000 links.

4.5.2 Accuracy

The traffic matrix impacts the effectiveness of our clustering algorithm and, hence, *Velo*'s accuracy. We evaluate *Velo* on real interdomain traffic matrices to show its accuracy in realistic settings. Also, we run *Velo* on synthetic traffic matrices to assess its sensitivity to different traffic characteristics.

We measure both the error bound ε as defined and the approximation error δ, i.e., the maximum difference in worst-case link loads between the approximated traffic matrix and the original one. We find that the approximation error is typically around 10× smaller than the error bounds. Our bound indeed *sums the absolute value* of all the clustering errors and assumes that all this traffic traverses every single link. Fortunately, the effect on the worst-case load can be positive and negative—their effects tend to cancel out, reducing the approximation error.

We compare our clustering algorithm to the basic approach of picking the destinations carrying the most traffic, which we denote as *Top-X*. To compare Top-X with *Velo*, we compute how many destinations are needed by Top-X to provide the same guarantees as *Velo*. That is, we find the smallest X such that the amount of traffic disregarded by Top-X equals our error bound ε. We then report the ratio between this X and $|C|$, as it directly measures the reduction of *Velo*'s running time.

Velo's running time effectively scales linearly with the number of clusters.

We relate real and synthetic traffic matrices to the gravity model, a common technique in traffic synthesis [147]. This model parametrizes the traffic matrix by three distributions:

$$m_{s,d} \propto \mathit{attraction}(d) \cdot \mathit{repulsion}(s) \cdot \mathit{friction}(s, d).$$

The *attraction* describes the traffic distribution towards the destinations: a heavy-tailed attraction results in a few destinations attracting the most traffic. The *repulsion* describes the distribution of traffic originating from each ingress symmetrically to the attraction. The *friction* describes traffic similarity across source-destination pairs: heavily tailed friction yields a sparse matrix where many entries are (essentially) zero.

Related work shows that all three distributions tend to be heavy-tailed in practice [144]. Our real data confirm this: the attraction, repulsion, and friction can be approximated as *LogNormal* distributions. We, therefore, characterize every traffic matrix according to the standard deviation σ of the *LogNormal* distributions representing attraction, repulsion, and friction. To this end, we compute the best-fit value of σ_{att}, assuming that the traffic for each destination $d \in P$ is sampled from a *LogNormal* distribution: $|\vec{\boldsymbol{m}}_d| \sim LogNormal(\sigma_{att}, \mu_{att})$. We do the same to characterize the repulsion, but focusing on the traffic that enters the network at each ingress $s \in N_{in}$, i.e., $|\vec{\boldsymbol{m}}_s| \sim LogNormal(\sigma_{rep}, \mu_{rep})$. For the friction between ingress $s \in N_{in}$ and destination $d \in P$, we divide the traffic $m_{d,s}$ from s to d by the repulsion $|\vec{\boldsymbol{m}}_s|$ of s and the attraction $|\vec{\boldsymbol{m}}_d|$ of d:

σ indeed measures the skewness of a LogNormal distribution.

$$\frac{m_{d,s}}{|\vec{\boldsymbol{m}}_s| \cdot |\vec{\boldsymbol{m}}_d|} \sim LogNormal(\sigma_{fri}, \mu_{fri}).$$

In all these experiments, we pick $|C| = 300$ clusters, as we find this to be a good tradeoff between accuracy and running time, but we evaluate different values of $|C|$ on Page 67.

Real traffic data. We obtain Netflow data for four 5-minute slices from the Swiss research network *Switch*. For each ingress and egress link, these data provide byte counters per destination IP, aggregated by /24 IPv4 or /48 IPv6 prefixes.

The data is extremely skewed: 0.23% of destinations attract 95% of all traffic. We stress that this does not seem to be common in the Internet [144, 148]. For example, 15% of destinations are reported to account for 95% of the traffic in the real traffic matrices studied in [144]. This difference may be an artifact of the destination granularityor due to the specificities of the research network originating our data.

We cannot reconstruct traffic for prefixes less specific than /24.

To protect users' privacy, all IP addresses are anonymized. We collaborate with *Switch* to create a mapping from IP prefix to their internal customers. Unfortunately, this mapping is neither unique nor complete: for operational reasons, some prefixes are related to multiple customers while others could not be assigned to customers, affecting around 30% of the total traffic volume. The known IP prefixes are then mapped to the associated (set of) customers before the anonymization step. We randomly assign the resulting anonymized identifiers to *Switch*'s customers. We analyze 100 such random assignments to confirm that this has little impact on the results.

real data	σ_{rep}	σ_{att}	σ_{fri}	error bound ε (0 – **10%**)	approx. error δ (0 – **1%**)	efficiency w.r.t. Top-X (1 – **60×**)
yes	**2.79**	**2.99**	**3.99**	5.63%	0.68%	4.3×
yes	**2.82**	**2.93**	**3.96**	5.34%	0.68%	3.7×
yes	**2.79**	**3.05**	**4.03**	5.07%	0.49%	4.2×
yes	**3.16**	**2.91**	**4.21**	3.46%	0.44%	3.7×
no	**2.03**	3.01	4.02	11.04%	1.06%	10.8×
no	**2.53**	3.01	4.03	8.79%	0.82%	15×
no	**3.00**	3.00	4.01	7.34%	0.88%	24×
no	**3.47**	2.99	4.01	6.47%	0.77%	40.4×
no	2.97	**1.98**	4.02	9.1%	0.89%	63.7×
no	2.98	**2.50**	4.01	8.52%	0.88%	39.1×
no	3.00	**3.00**	4.01	7.34%	0.88%	24×
no	3.02	**3.51**	4.01	6.01%	0.72%	14.8×
no	3.04	**4.01**	4.01	4.58%	0.52%	8.9×
no	2.95	3.00	**2.51**	12.18%	0.88%	48.6×
no	2.96	3.01	**3.01**	11.07%	0.91%	36.8×
no	2.97	3.00	**3.50**	9.43%	0.85%	30.4×
no	3.00	3.00	**4.01**	7.34%	0.88%	24×
no	3.01	3.00	**4.51**	5.85%	0.67%	18.9×

Table 4.2: *Comparison of the error bound ε and approximation error δ for different traffic matrices. Boxes represent the 25 and 75 percentiles, with the mean indicated by the red line. Whiskers show the minimum and maximum values. The number right to the upper whisker shows the maximum number.*

The first four rows in Table 4.2 show *Velo*'s accuracy for all such traffic matrices. Our system guarantees an error bound ε of around 5% and has an actual approximation error δ of around 0.5%. Further, *Velo* is $\approx 3.4\times$ more efficient than top-X, despite the extreme traffic skewness that favors Top-X.

Synthetic data. Based on our real traffic data, we synthesize matrices with similar characteristics using the gravity model. Starting from the average parameters of the real matrices, we evaluate how changing the skewness in all three dimensions affects *Velo*'s accuracy, one at a time. We do so for six large topologies from TopologyZoo with 140–200 nodes. For each topology and set of parameters, we sample 20 random matrices.

Table 4.2 demonstrates the effectiveness of *Velo*'s clustering for a wide range of traffic patterns. The error bound ε is mostly below 10%, and the actual approximation error δ is always <1%. As expected, decreasing traffic skewness (for any parameter) increases both ε and δ because it causes traffic to be more evenly spread across sources and destinations and, hence, harder to capture with a fixed number of clusters. Yet, $|C| = 300$ clusters accurately approximate unrealistic traffic matrices where traffic distributions are not very skewed at all.

number of clusters	time	error bound ε	approx. error δ
100	178 s	16.23%	2.66%
300	210 s	7.34%	0.88%
600	268 s	4.95%	0.39%
1000	343 s	3.71%	0.22%

***Table** 4.3: This table lists the accuracy of Velo in terms of the error bound and the approximation error for different numbers of clusters, together with the analysis time.*

Clustering is also 5x-50x more efficient than Top-X. In fact, Top-X needs a lot more destinations to achieve the same error bound as *Velo*, especially if the attraction has a less significant tail. We conclude that *Velo*'s clustering is more effective and more general than Top-X.

Performance–Accuracy Tradeoff. Intuitively, increasing the number of clusters $|C|$ used by *Velo* increases its accuracy at the cost of longer running times, as we also evaluate in Section 4.5.1. This is because more clusters provide a more accurate view of the original traffic matrix but also prevent *Velo* from reusing results, hence requiring more computations.

We now evaluate the empirical impact of the number of clusters $|C|$ on the performance and accuracy. We run *Velo* on the same six large topologies (with 140–200 nodes) with synthetic traffic matrices similar to our real ones and repeat each experiment 20 times. Table 4.3 shows *Velo*'s running time and accuracy for 100, 300, 600, and 1 000 clusters for $l = 2$ and $k = 10$. We identify a substantial improvement in accuracy when increasing $|C|$ from 100 to 300, especially in terms of the error bounds. However, increasing $|C|$ to 1000 yields smaller improvements at the expense of a 60% increase in running time.

We conclude that $|C| = 300$ provides a reasonable tradeoff between accuracy and running time. This is why we use this value in most of our evaluations. However, we stress that the number of clusters is configurable in *Velo*, so they can be tuned by the operators according to their needs to either reduce the running time or improve the accuracy.

Traffic Variability. We finally evaluate *Velo*'s accuracy when traffic volumes additional to the traffic matrix are given as input, cf. Section 4.1. Across all experiments described in Figure 4.5, *Velo* computes the correct maximum link loads in over 99.9% of all links for any value of additional traffic.

Takeaways. *Velo* is very accurate in approximating the worst-case link loads, even when using only $|C| = 300$ clusters. In all our experiments, *Velo*'s approximation error is below 1% of the total traffic volume. Further, *Velo* guarantees error bounds below 12%, even for unrealistic traffic matrices that are challenging to approximate.

4.6 Case Study

We use *Velo* to analyze the topology and configuration of *Switch*, a Swiss research network that connects hundreds of institutions. We rely on the same traffic matrices described in Section 4.5.2 and *Switch* topology, link capacities, and configuration at the time the traffic matrices were recorded.

Switch observes 60% of its traffic towards their customers, i.e., ingress traffic, and 40% from their customers to remote destinations, i.e., egress traffic. In this case study, we focus on both ingress for ingress traffic and egress changes for egress traffic. This case study complements the one in [2], which focuses on egress changes in transit (non-residential) ISPs.

No Switch customer is multihoming through a third-party ISP; they do not carry transit traffic. Further, they only sample packets on links to/from external peers. They cannot measure the traffic that is exchanged between their customers, i.e., internal traffic.

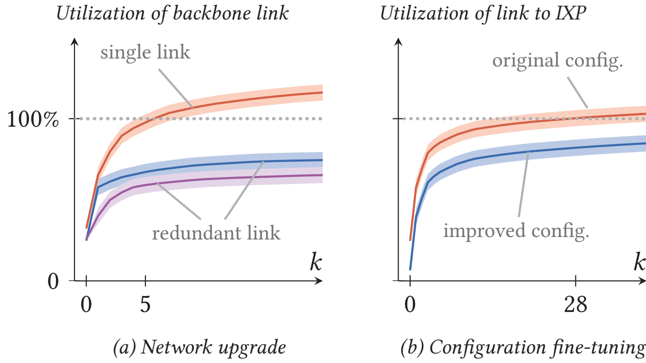
Then, we run *Velo* on the relevant parts of their router configurations with an increasing number k of routing changes. Within a few seconds, *Velo* finds that the input configurations and topology are fragile to specific routing changes: two links approach their capacity with a few routing changes and get congested at $k = 5$ and $k = 28$, respectively.

We now discuss how the insights from *Velo*'s analyses can help operators optimize their networks.

Aiding strategic decisions. The first fragile link connects *Switch* to an IXP. The link will be congested if traffic towards just five customers enters *Switch* at that IXP.

Upgrading the fragile link may seem the obvious solution. However, *Velo*'s output highlights that other links in *Switch* would also need to be upgraded to sustain the additional traffic.

A better alternative is to add a link from the IXP to another geographically close router in the *Switch* backbone. Re-running *Velo* on the alternative topology indeed reveals that no link would be congested in this case. Figure 4.7a compares the robustness of the initial and alternative topologies.



(a) Network upgrade *(b) Configuration fine-tuning*

Figure 4.7: *Our case study demonstrates that Velo helps to improve network robustness.*

Robustifying configurations. The second fragile link is part of the *Switch* backbone. The link would be congested by specific routing changes for 28 destinations.

Since operators are likely unwilling to upgrade the topology in this case, we use *Velo* to find a more robust *configuration.* We identify five MPLS paths that would effectively shift some traffic to underutilized links. Running *Velo* on such an alternative configuration ensures that the link utilization in the entire network remains below 90% even after 50 routing changes, as displayed in Figure 4.7b.

Improving network management. *Velo* identifies 25 pairs of prefixes and ingress or egress links. Link loads significantly increase if traffic either enters or leaves the network at the corresponding link. This information is highly valuable for network management and operation. For example, it enables monitoring systems to focus only on those 25 pairs of prefixes and ingress or egress routers, alerting operators only for the small subset of critical events that require immediate action.

Velo's output can also improve automated management systems. For example, it enables traffic engineering systems (e.g., [80, 81, 149]) to re-optimize paths *before* any link load requirement is violated. After observing a few routing changes of those 25 mentioned above, we can trigger an emergency recomputation of internal paths, thus *preventing* congestion without waiting for data-plane measurements that are typically collected only every few minutes [80]

4.7 Related Work

Velo focuses on verifying link load properties, e.g., helping to improve network robustness to external events. As such, it is related to prior work in network verification and routing.

Network verification. Most control-plane verification systems ([48, 49, 62, 73, 75], just to name a few) focus on properties such as forwarding and propagation paths, which cannot directly be translated to link loads. These systems cannot be easily extended to verify link load properties because they verify one destination at a time. As failures break the independence of destinations, thousands of destinations must be considered jointly for link load properties. Doing so with prior approaches would simply not scale: for techniques like [66], it would inflate SMT formulas by $|P|\times$ with $|P|$ destinations.

The few existing checkers of link load properties *do not support BGP route changes. Jingubang* [131] is a flow-level simulator that computes link loads for specific failure scenarios and traffic matrices. *YU* [70] extends this simulator to deal with k arbitrary link failures: it still works at the per-flow level. Our closest related work is *QARC* [69] that verifies link load *violations* (without computing all link loads) upon link failures. We experimentally compare *Velo* against *QARC* in Section 4.5.1.

Routing systems. Optimizing link loads has been the focus of a large body of work [79, 150, 151], especially in the area of traffic engineering (TE). TE approaches typically optimize paths based on the *current* traffic and BGP routes or, at best, variations of them [130, 152, 153]. Some of these approaches [132, 154, 155] rely on robust optimization to *approximate* the impact of failures. Future work may try to apply robust optimization techniques for link-load verification, although it is unclear how these techniques would guarantee accuracy and scalability when analyzing both failures and route changes.

Real-world TE systems [80, 81] update forwarding paths upon detecting a failure and react to BGP route changes by re-optimizing paths according to load measurements collected every few minutes. By anticipating the most impactful failures and route changes, *Velo* can be integrated into TE systems to prevent excessive link loads or react faster to them, see Section 4.6.

Information on current and future traffic is key for intradomain traffic engineering. Recently, systems, e.g., [156], have been proposed to predict changes in incoming traffic caused by specific BGP routes *announced to* external neighbors. We envision that such systems can be used by operators to compute traffic matrices they want to verify with *Velo*.

4.8 Discussion

In this section, we discuss some of the design choices we made in *Velo* alongside the generality of the approach.

Why not *modeling congestion?.* For each link, *Velo* computes the load by simply summing the amount of traffic routed over it. In practice, though, no link load could exceed its capacity: traffic would be dropped instead. This also means that the amount of traffic forwarded through the network immediately decreases for the links downstream of the congested one. Hence, when any link is congested, the link loads reported by *Velo* do not reflect the actual traffic volumes on links.

Despite this simplification, *Velo*'s approach is practical for at least two reasons. First, *Velo* correctly identifies *all* the links that cannot sustain the input traffic. Second, it correctly finds the amount of exceeding traffic, and it therefore helps quantifying how much traffic needs to be rerouted, or by how much links' capacity should be upgraded.

What protocols and features can Velo model?. Despite hiding most of the complexity of BGP and the intradomain routing protocols, *Velo*'s model is powerful enough to capture common routing architectures, including iBGP and SDN (see Section 4.2). In fact, our model only imposes two constraints on intradomain routing. First, all routers must be able to select any of the egress routers, i.e., there are no iBGP visibility problems. There exists several techniques to avoid these problems [136, 138, 139]. Second, destinations must be independent, which requires the de-activation of intradomain routing features that tie different destinations together, such as route aggregation or conditional advertisements. The use of these features is discouraged in iBGP [37]. Additionally, we think that *Velo* can be extended to jointly reason about egresses of dependent destinations.

Data-plane filters like ACLs do not influence routing, and thus, Theorem 4.2 still applies. Algorithms 4.2 and 4.4 can be easily extended to apply such filters by dropping (some) traffic instead of propagating it. Doing so would ensure that the worst-case link loads computed by *Velo* would be consistent with both routing inputs and data-plane filters.

What routing changes does Velo support?. *Velo* computes the worst-case link loads for routing changes affecting subsets of destinations. In Section 4.2, we have defined destinations as the IP prefixes in the routing tables. This allows *Velo* to cover BGP routing changes for any of these IP prefixes.

In reality, the set of IP prefixes learned via BGP can change over time. For example, at any time, the remote network owning an IP prefix, say 1.0.0.0/8, can start announcing a more specific prefix, such as 1.0.0.0/16. If this happens, BGP propagates two distinct and independent routes – e.g., one for 1.0.0.0/8 and the other for 1.0.0.0/16.

Since *Velo* learns the destinations from the traffic matrix, it supports some analyses of routing changes affecting the set of known destinations, provided that the operators provide enough information. For instance, *Velo* covers the example above if the input matrix includes traffic volumes for 1.0.0.0/16 and (separately) for the other subnets in 1.0.0.0/8. In any case, this information is necessary to compute link loads with distinct routes for 1.0.0.0/16 and 1.0.0.0/8.

In contrast, *Velo* currently does not support a systematic explorations of routing changes affecting the set of destinations, e.g., announcements of *any* possible sub-prefix of a current destination. We leave this direction for future work.

4.9 Conclusions

Velo demonstrates the feasibility of verifying worst-case link loads in large networks despite the need to navigate a gigantic space of possible failures and route changes. Our evaluation shows that *Velo* is both scalable and accurate across a wide range of topologies, settings, constraints, and traffic matrices.

Velo focuses on maximum link loads, as they have a straightforward practical relevance to assess network performance. As our case study demonstrates, *Velo* readily supports operators

in complex tasks, ranging from improving router configuration to network design and aiding business decisions.

We see *Velo* as a stepping stone towards the broader goal of verifying network performance. We believe that exploring other performance aspects, such as delay or bandwidth guarantees, is an interesting avenue for future research.

Chapters 3 and 4 focus on network verification, proving that a specification is satisfied in a wide range of environments and, if not, finding one that violates the specification. In the following two chapters, we shift our attention to the other kinds of network management tools: seamlessly reconfiguring networks and synthesizing configurations to satisfy a high-level intent.

5 *Taming the transient while reconfiguring BGP*

Much has been written about network reconfigurations, their frequency [5, 25, 157–159], and their disruptiveness [5, 93, 160]. Yet, reconfiguration-induced downtimes *still* happen. In fact, Alibaba recently revealed that the *majority* of their network outages resulted from configuration updates [160].

Among all reconfiguration scenarios, BGP ones are special because they are both particularly frequent and potentially highly disruptive. In large networks, for instance, operators reconfigure BGP up to 20 times a day on average [25]. Also, since BGP controls routing to (and from) remote destinations, reconfiguring it can have Internet-wide consequences. The recent Microsoft outage in January 2023 perfectly illustrates this: Microsoft Azure services were indeed unavailable for *90 minutes* due to a BGP reconfiguration [161].

Perhaps surprisingly, no existing network reconfiguration framework enables both *safe* (in a way that preserves network invariants) and *practical* (in a way that works operationally) BGP reconfigurations. Most previous works targeted other reconfiguration scenarios, such as networks running purely intradomain routing protocols (OSPF or IS-IS) or SDN/OpenFlow [96]. A few techniques [5, 25, 104] focus on reconfiguring BGP, but they suffer from fundamental limitations. Specifically, "Shadow Configurations" [104] and BGP Ships-in-the-Night [25] perform the reconfiguration by duplicating both the routing and forwarding states on every network device. Doing so is impractical, though, as it comes with significant overhead and is not supported by most routers [96]. In contrast, *Snowcap* [5] gradually modifies BGP configurations in place while ensuring that the *converged* network states are correct. *Snowcap* does *not* guarantee the correctness of any of the transient states explored as the network converges during the reconfiguration.

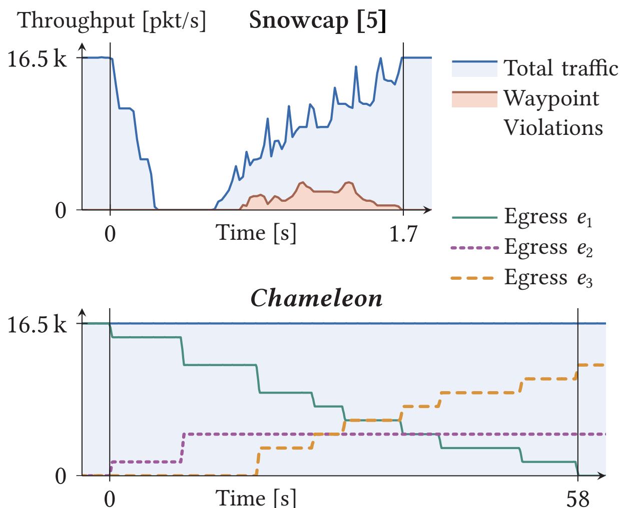


Figure 5.1: *BGP reconfigurations often lead to disruptions, even with reconfiguration systems such as Snowcap [5]. Here, Snowcap transiently violates two invariants: reachability and waypointing. In contrast, Chameleon reconfigures the network without violating any.*

The network receives routes from three neighbors at egress e_1, e_2, and e_3, preferring the one from e_1. The reconfiguration involves the eBGP session at e_1, causing all routers to switch to e_2 or e_3.

We illustrate the limitations of *Snowcap* in Figure 5.1, which depicts the evolution of the throughput during a simple BGP reconfiguration of the *Abilene* network. Throughout the entire reconfiguration, the network must maintain two invariants: *(i)* reachability, i.e., no traffic should ever be dropped, and *(ii)* waypointing, i.e., traffic should always cross a firewall. Yet, *Snowcap* transiently violates both for almost two seconds. For critical requirements, such as those that relate to SLAs or to security, transient violations are problematic. We repeat these experiments for different scenarios, which show a similar trend and even longer violations in some cases, cf. Figure 5.7.

The reconfiguration in Figure 5.1 involves a single command. Hence, systems like Snowcap that find a safe ordering of the reconfiguration commands cannot safely perform such a reconfiguration.

In this chapter, we present *Chameleon*, the first *in-place* BGP reconfiguration framework that guarantees correctness during the *entire* reconfiguration process for both the steady *and* all transient states during convergence.

Intuitively, building a framework like *Chameleon* requires solving a concurrency problem. It indeed entails reasoning about the asynchronous computations and non-deterministic message exchanges of all BGP routers in the network. This is challenging for at least three reasons. First, reasoning about transient states necessitates a formal concurrency model of a converging BGP network. Yet, existing network verification models only reason about properties in the stable state [38, 108] or whether such a stable state will be reached eventually [107].

Second, safely reconfiguring BGP requires concurrency control mechanisms, i.e., synchronization techniques, that *provably* maintain network-wide correctness without controlling the timings of individual BGP messages. Finally, it requires the design of a runtime controller that can efficiently orchestrate the entire reconfiguration process.

We address the challenges in the following ways. First, we introduce a happens-before model that captures BGP-specific concurrency. This model serves as a tool to constrain BGP route propagation with synchronization barriers while provably maintaining invariants for all possible concurrent executions. We compile this schedule into a reconfiguration plan that enforces the computed route propagation by slightly tweaking route preferences while ensuring synchronization using inexpensive local checks. Finally, our runtime controller applies this reconfiguration plan to the live network by pushing standard BGP commands.

Ensuring correctness throughout the entire reconfiguration process does not come for free: the need to coordinate some of the BGP computation means that the reconfiguration will typically last longer. This cost is exemplified in Figure 5.1, where *Chameleon* took 58 sec to complete instead of 1.7 sec for *Snowcap*. All in all, we think that such an increase in duration is a small price to pay for the increased correctness, especially when the alternatives are either duplicating the entire control plane to use the Ships-In-The-Night technique [25] or violating SLAs or critical security properties.

As we explain in Section 5.5, most of the time is spent waiting for routers to start applying changes to route preferences, presumably in anticipation of additional changes to apply them in bulk. The actual convergence time, i.e., the time to reach the synchronization barriers, amounts to only a few seconds.

We implement *Chameleon* and evaluate it in both testbeds and simulations on real-world topologies and large and realistic reconfiguration scenarios. In most experiments, *Chameleon* computes reconfiguration plans in <1 min and performs them from start to finish in a few minutes, with minimal overhead.

To sum up, our main contributions are:

- The first practical BGP reconfiguration technique capable of preserving correctness invariants throughout the entire reconfiguration campaign, including transient states.
- A concurrency model for BGP reconfigurations and its convergence process using happens-before relations.
- A complete implementation of *Chameleon* in Rust alongside an online application to explore reconfiguration scenarios (available at *bgpsim.github.io?s=example*).

Temporal operators				**Logical operators**			
ϕ	::=	ϕ	*now*	ϕ	::=	$\phi \land \phi$	*conjunction*
	|	$\mathbf{N}\,\phi$	*next*		|	$\phi \lor \phi$	*disjunction*
	|	$\mathbf{G}\,\phi$	*globally*		|	$\neg\phi$	*negation*
	|	$\mathbf{F}\,\phi$	*finally*				
	|	$\phi\,\mathbf{U}\,\phi$	*until*	**Propositional variables**			
	|	$\phi\,\mathbf{R}\,\phi$	*release*	ϕ	::=	$reach(n)$	*reachability*
	|	$\phi\,\mathbf{W}\,\phi$	*weak **U***		|	$wp(n, n)$	*waypointing*
	|	$\phi\,\mathbf{M}\,\phi$	*mighty **W***	n	::=	*node*	*node*

***Figure** 5.2: Chameleon's grammar for constructing a specification ϕ for a single destination. We ensure that ϕ is satisfied during the entire reconfiguration, i.e., the sequence of forwarding states satisfies ϕ.*

5.1 Overview

We now provide an overview of *Chameleon*. We start with the problem statement and illustrate the challenges using a running example before describing *Chameleon*'s high-level workflow.

5.1.1 Problem Statement

BGP Reconfiguration. Changing the BGP configuration of one or more routers may involve *(i) local* changes like adding or removing a BGP session or changing the local preference of some specific routes and *(ii) global* changes such as switching from an iBGP full-mesh to route reflection network-wide.

Specification. We refer to a *specification* ϕ as a set of network invariants that describe forwarding properties the operator wants to maintain. Our specification language (cf. Figure 5.2) combines individual properties with boolean operators and Linear Temporal Logic (LTL). Boolean operators allow users to express intricate properties on the forwarding state, such as isolating traffic from parts of the network. In contrast, temporal operators formulate constraints on the sequences of transient states explored during the reconfiguration. For example, if a reconfiguration changes the egress router of a given destination prefix, operators can require routers to switch from the initial to the final egress router *once*, without switching back and forth multiple times, which can deteriorate the quality of service.

We assume that both the initial and final configurations are correct and comply with ϕ. Otherwise, there is little point in maintaining correctness *during* the reconfiguration. Similarly, we assume that the initial and final configurations eventually converge to a stable state [107, 135] to ensure that the semantics of input specifications are always sound.

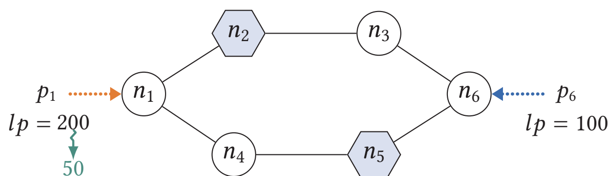


Figure 5.3: *Example network with six internal routers. Two routers n_1 and n_6 receive a route p_1 and p_6, respectively, for the same prefix. The depicted reconfiguration lowers the local preference of p_1 from 200 to 50, causing the network to shift from using p_1 to p_6.*

Goals. We aim at performing both local and global BGP reconfigurations in-place while guaranteeing the correctness of a given specification ϕ throughout the *entire* process. Specifically, we aim to achieve the following two properties:

- *Safety:* The reconfiguration process must guarantee that the input specification ϕ is satisfied throughout any transient state explored during the BGP convergence process.
- *Practicality:* The reconfiguration process must limit the overhead it imposes on routers' resources, including routing and forwarding table sizes and reconfiguration time.

Running example. As an illustrative reconfiguration scenario, we consider the network in Figure 5.3, which consists of six BGP routers, two of which (n_2 and n_5) operate as route reflectors [39]. Routers n_1 and n_6 each receive one BGP route *for the same prefix*. We refer to those routes as p_1 and p_6, respectively. The reconfiguration involves decreasing the local preference of p_1 from 200 to 50 so that all routers switch to using p_6. We suppose the operators aim at preserving reachability throughout the reconfiguration, which they can express as $\phi = \mathbf{G} \bigwedge_i \mathit{reach}(n_i)$ in our specification language.

Route reflectors distribute their best BGP routes they receive to other routers in the network.

Preferring p_6 will cause all routers to forward "to the right" instead of "to the left." This causes a loss of reachability, i.e., violating the specification, whenever routers on the left-hand side of the network update their state before routers on the right. Depending on the timing of the BGP messages, this can easily happen: Say n_6 gets notified about the lower local-reference or p_1, thus preferring p_6. Then, n_6 will propagate p_6 to both n_2 and n_5, the two route reflectors. Receiving this new route, n_2 will pick p_6 and immediately start forwarding towards n_6 while n_3 still prefers p_1, triggering a transient forwarding loop.

Intuitively, the reconfiguration could be performed without transient forwarding anomalies by updating routers from the right to the left. Indeed, *Chameleon* ensures that n_3 updates its forwarding before n_2 through temporary BGP commands.

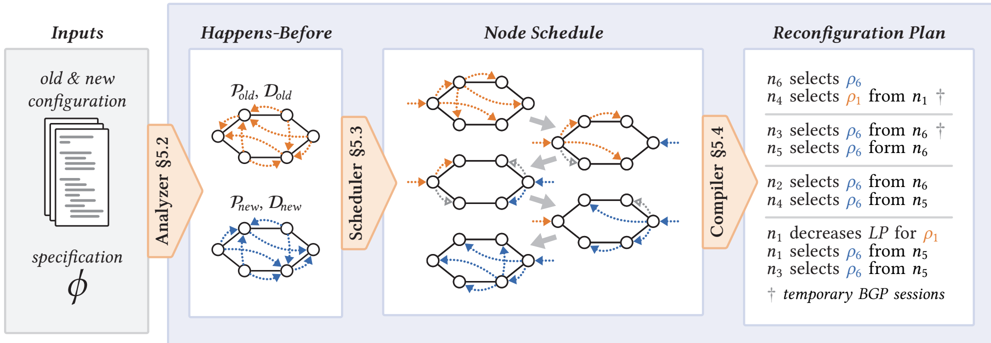


Figure 5.4: Chameleon computes a reconfiguration plan to transition to the new configuration in a way that satisfies the specification. This scenario corresponds to the example from Figure 5.3. Explore this reconfiguration plan interactively at bgpsim.github.io?s=example.

5.1.2 Chameleon

In a nutshell, *Chameleon* achieves safety by coordinating the forwarding state updates that occur during the reconfiguration. To coordinate the updates practically, *Chameleon* gradually introduces a temporary BGP configuration, which differs from both the initial and final one. This temporary configuration enforces a particular and safe ordering of BGP messages.

Finding and implementing specific message orderings is far from trivial. The first challenge is capturing which orderings are achievable by introducing temporary BGP configurations. This is hard as BGP-specific mechanisms like route reflection limit route visibility [135]. The second challenge is enforcing a specific execution using temporary configurations; we must reason about the distributed routing state and synchronize BGP computations at the same time.

Workflow. *Chameleon* solves those challenges by following three consecutive steps. *Chameleon*'s workflow is visualized in Figure 5.4 using the running example from Figure 5.3.

1. The *analyzer* describes the space of concurrent convergence processes by analyzing the initial and final configuration (the input to *Chameleon*) and computing happens-before relations between routing states of different routers.

2. The *scheduler* explores the space of convergence processes spanned by the happens-before relations to find one that satisfies the specification. It describes this convergence process as a node schedule that captures which routes are selected at which time.

3. The *compiler* transforms this node schedule into a detailed *reconfiguration plan*, that is, a sequence of configuration commands and local conditions needed for synchronization.

Finally, our runtime controller reconfigures the live network by checking the local conditions and applying the commands, precisely following the compiled reconfiguration plan. In the following, we describe all three steps of *Chameleon* in more detail, using the running example from Figure 5.3.

Step 1: Analyzer §5.2. The analyzer extracts happens-before relations between selected routes in the network, encoding the propagation path of routes. These relations define the space of convergence processes that are realizable just using temporary BGP configurations. We obtain these relations by simulating the network and analyzing the resulting network state in a way that generalizes to any BGP configuration.

In our example from Figure 5.3, router n_1 propagates the initial route p_1 to both route reflectors n_2 and n_5, who then announce p_1 towards n_3, n_4, and n_6. Therefore, n_4 can choose p_1 as long as n_2 or n_5 select p_1. We repeat the same for p_6.

Step 2: Scheduler §5.3. We formulate these happens-before relations together with the specification as an ILP. A solution to the ILP is a node schedule that describes a specific BGP convergence process that satisfies the specification in both steady and transient states. The scheduler allows concurrent updates if their relative order does not affect the specification. Further, the scheduler may need to establish additional propagation paths to increase route visibility and ensure a solution exists. As a primary objective, we maximize the number of concurrent updates, which also minimizes the reconfiguration time, while reducing additional propagation paths as a secondary goal to further reduce the overhead.

Chameleon schedules the example reconfiguration using four rounds, updating nodes from right to left. It also computes the rounds in which a router selects its old or new route. In Figure 5.4, the orange arrows represent the propagation of the old route p_1, whereas the blue arrows depict the new one p_6. *Chameleon* can reconfigure the network in four rounds only by introducing two temporary propagation paths as shown in gray. Otherwise, it would take six rounds, updating each router individually.

Some scenarios necessitate using temporary propagation paths. This is, for instance, the case when only n_2 is configured as a route reflector, further reducing route visibility.

Step 3: Compiler §5.4. *Chameleon* generates a reconfiguration plan to implement the calculated schedule, along with local conditions, to synchronize the update and guarantee correctness. Each temporary command only rewrites routing preferences by modifying route attributes such as weight [162] or local preference. Likewise, the conditions assert a router knows or selects a specific route. These can be tested locally by inspecting that router's Routing Information Base (RIB).

Chameleon generates a reconfiguration plan for our example that first updates the egress router n_6 to make p_6 available in the network. In doing so, the preference of p_6 remains low or all routers apart from n_6, so they do not *yet* select it uncontrollably. It then makes both n_5 and n_3 select p_6 using a temporary session between n_3 and n_6. In Round 3, it updates both n_2 and n_4. However, even though n_2 already knows p_6, it can only select it once n_3 has selected it as well—a synchronization barrier. *Chameleon* introduces a temporary session between n_4 and n_1 to break n_4's dependency on n_2. Finally, *Chameleon* updates the local-pref on n_1, which causes n_1 to select p_6. Now, all routers choose p_6, and p_1 is no longer available, and *Chameleon* can safely remove any temporary configuration.

n_4 would switch uncontrollably if we did not add this session.

Updating multiple destinations. If the reconfiguration updates multiple destinations at a time, *Chameleon* runs the scheduler and the compiler for each destination separately and combines the generated schedules. This is possible if all prefixes converge independently, which is valid for most networks. However, there exist networks in which some prefixes are dependent. We explain in Section 5.7 how the scheduler can be extended to support such networks.

The remainder of this chapter is structured as follows: We first explain all three steps of *Chameleon*, that is, the Analyzer, Scheduler, and Compiler, in Sections 5.2–5.4. In Section 5.5, we then present a case study in which *Chameleon* reconfigures physical networks. In Section 5.6, we evaluate *Chameleon*'s overhead by testing it in a simulated setting on a wide range of topologies, reconfiguration scenarios, and specifications while measuring its scheduling and reconfiguration time, as well as the necessary routing table size. Finally, we discuss the limitations of *Chameleon* in Section 5.7, present related work in Section 5.8, and conclude this chapter in Section 5.9.

5.2 *Analyzer*

Chameleon models the routing state of BGP as propagation paths of BGP routes. We describe the reconfiguration process in terms of how these change over time and formulate happens-before relations capturing safety conditions on BGP transient states. While *Chameleon* is designed with BGP in mind, the presented approach can be generalized to many distance-vector protocols.

Network & Routing Model. We model the network as a graph $G = (N, E)$ with nodes N connected by edges E. External peers advertise BGP routes towards a destination d to multiple nodes, the egress routers of G. Nodes propagate routes over BGP sessions that form an overlay signaling graph, which is different from the physical topology. We define routes in terms of their *propagation path* rather than modeling a route's specific BGP attributes. If $p = \langle d, n_1, \dots, n_i, n \rangle$ is a route received at node n, then p is advertised by $src(p) = d$, first received by the egress router $egress(p) = n_1$, and advertised to n by its BGP neighbor $n_i = from(p)$. The network's *configuration* determines:

In contrast to the model of the BGP State Iterator from Chapter 3, a propagation path here denotes the sequence of routers and not the sequence of edges.

1. which route p a node n selects,
2. to which neighbors n announces its selected route p,
3. how the $egress(p)$ is mapped to the next hop.

We call the collection of next hops for all nodes a *forwarding state*. For a single destination d, we express the forwarding state $nh : N \mapsto N \cup \{d, \emptyset\}$ as a mapping of each node to its next hop. Node n drops packets if $nh(n) = \emptyset$, and forwards them to the external network if $nh(n) = d$.

A *routing state* $\mathcal{P}$ assigns each node n its selected route $\mathcal{P}(n)$ for destination d. We call a routing state $\mathcal{P}$ *consistent* if, for all nodes $n \in N$, the neighbor $from(\mathcal{P}(n))$ selects the prefix of the route $\mathcal{P}(n)$. More formally,

$$\mathcal{P}(n) = \langle d, n_1, \dots, n_i, n \rangle \Rightarrow \mathcal{P}(n_i) = \langle d, n_1, \dots, n_i \rangle.$$

Observe that the routing state of any *converged* network, that is, a network in a fixed routing state with no more messages in flight, is always consistent. However, the consistency of a routing state only ensures that a configuration (including corresponding routing advertisements from external networks) exists that the network converges to this routing state.

Using these terms, we define a *BGP reconfiguration* as the transition from an initial routing state $\mathcal{P}_{old}$ to the final routing state $\mathcal{P}_{new}$. A *reconfiguration process* $[\mathcal{P}_{old}, \mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_{new}]$ is a sequence of routing states that contains all the intermediate routing states of the network during a reconfiguration. We call a reconfiguration process *safe* if its corresponding sequence of forwarding states satisfies the specification that the operator expresses $[nh_{old}, nh_1, \dots, nh_{new}] \models \phi$.

Our routing model considers a single destination d at a time. However, one destination can represent an entire class of equivalent prefixes for which the network computes the same routing and forwarding state. If a BGP reconfiguration affects multiple destinations, *Chameleon* treats each prefix equivalence class separately: this is safe because BGP separately processes individual destinations.

Methodology & framework. *Chameleon* precisely controls intermediate routing states during the reconfiguration to safely transition from $\mathcal{P}_{old}$ to $\mathcal{P}_{new}$. To that end, *Chameleon* synthesizes a sequence of routing states so that each state is consistent. Consequently, the BGP convergence process never affects the forwarding state. Instead, *Chameleon* explicitly triggers each routing and forwarding state update by reconfiguring nodes.

A naive approach to synthesizing a sequence of routing state updates is to make each node n switch its selected route directly from the initial route $\mathcal{P}_{old}(n)$ to its final one $\mathcal{P}_{new}(n)$. This technique, however, does not work for most reconfiguration scenarios. Assume node n learns both its initial and final route from $n_i = from(\mathcal{P}_{old}(n)) = from(\mathcal{P}_{new}(n))$. The naive approach cannot ensure the consistency of every routing state, as n_i must switch before n. While n_i is using $\mathcal{P}_{new}(n)$ and n still selects $\mathcal{P}_{old}(n)$, the routing state is inconsistent. *Chameleon* uses two techniques to resolve this issue.

The first technique is to establish a temporary BGP session directly with the egress router, $egress(\mathcal{P}_{old}(n))$ or $egress(\mathcal{P}_{new}(n))$. This session allows n to select the route $\langle d, n_1, n \rangle$ directly. This technique has two significant downsides. First, the additional session adds overhead for maintaining this BGP session and likely causes state duplication. Second, the new route may have different BGP attributes that affect the route propagation. Thus, we must ensure no other node selects a route from n while n selects $\langle d, n_1, n \rangle$.

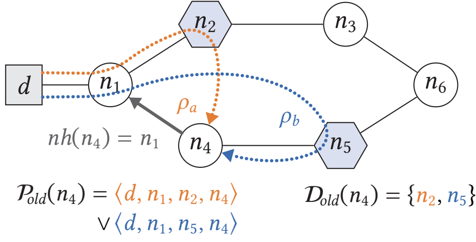


Figure 5.5: *Illustration of the BGP convergence model based on the example from Figure 5.3 in the initial state.*

Our second technique to ensure consistent routing states is to leverage BGP redundancy. Any router could potentially receive *equivalent* routes from multiple route reflectors that all share the same BGP attributes. *Chameleon* leverages this redundancy to gain more degrees of freedom when generating the reconfiguration plan. We use a setup and cleanup phase, during which *Chameleon* chooses between equivalent routes.

The setup and cleanup phase so ensures routers cannot unexpectedly switch between two equivalent routes, ensuring all routing states are consistent.

Our methodology results in each node n updating its next hop exactly once from $nh_{old}(n)$ to $nh_{new}(n)$. *Chameleon* never exports any transient routes towards interdomain neighbors.

There may exist highly constraining specifications that we cannot enforce. Yet, our methodology is guaranteed to perform any reconfiguration while preserving reachability. This holds because updating the next hop of nodes one by one is akin to the update problem in SDN, for which there always exists a node update schedule that preserves reachability [100].

Happens-before relations. We model a BGP reconfiguration process within our framework using happens-before relations. We capture the set of all neighbors that advertise equivalent routes as the initial and the final route towards node n using $\mathcal{D}_{old}(n)$ and $\mathcal{D}_{new}(n)$, respectively. n cannot use its initial route $\mathcal{P}_{old}(n)$ for longer than any of $\mathcal{D}_{old}(n)$ select their initial route. Likewise, n can only select $\mathcal{P}_{new}(n)$ as soon as any of $\mathcal{D}_{old}(n)$ select their final route. The resulting happens-before relations for node n are:

We rely on BGPSim to simulate both the initial and final state and extract $\mathcal{D}_{old}$, $\mathcal{P}_{old}$, $\mathcal{D}_{new}$, and $\mathcal{P}_{new}$.

- n selects $\mathcal{P}_{old}(n) \implies \exists m \in \mathcal{D}_{old}(n)$ that selects $\mathcal{P}_{old}(m)$.
- n selects $\mathcal{P}_{new}(n) \implies \exists m \in \mathcal{D}_{new}(n)$ that selects $\mathcal{P}_{new}(m)$.

Figure 5.5 illustrates the happens-before relations for the initial state using the running example introduced in Figure 5.3. Node n_4 receives equivalent routes for d with propagation paths p_a and p_b from route reflectors n_2 and n_5. Thus, n_4 can select route $\mathcal{P}_{old}(n_4)$ for as long as either n_2 or n_4 advertise it.

5.3 Scheduler

We now detail *Chameleon*'s scheduler. The goal of the scheduler is to generate a node schedule, that is, the order in which routers switch from selecting their old route in $\mathcal{P}_{old}$ to their new route $\mathcal{P}_{new}$. We encode this schedule by assigning each node a round in which it will change its routing (and forwarding) decision. We formalize the problem of finding a schedule as an ILP. Our ILP captures these three high-level properties:

1. *Happens-before relations*: We transform happens-before relations from Section 5.2 into constraints. We design the objective function to minimize the number of temporary BGP sessions.
2. *Concurrent Updates*: Allowing nodes to update concurrently reduces the number of required rounds. However, this concurrency can lead to different update orderings within a single round. Therefore, we require all routers to change their next hop in a round where *no* other router along its path also changes its forwarding decision. This constraint guarantees updates in the same round are *independent* concerning the forwarding state.
3. *Specification*: The resulting sequence of forwarding states must satisfy the specification ϕ, expressed as LTL.

Each ILP has a fixed number of rounds R. We increment R in a loop as long as no solution exists, thus minimizing the number of rounds R and, consequently, the reconfiguration time. For each generated ILP, we minimize the number of temporary BGP sessions necessary, thus forming the secondary objective function after minimizing R.

5.3.1 Happens-Before Relations

We enforce BGP propagation rules in the ILP model by using three symbolic integer variables for each node $n \in N$:

$$\bigwedge_{n \in N} r^n_{old} \leq r^n_{nh} \leq r^n_{new}. \tag{1}$$

We call the 3-tuple $\left(r^n_{old}, r^n_{nh}, r^n_{new}\right)$ the schedule of n. $r^n_{nh} \in \mathbb{N}$ describes the round in which n changes its next hop. $r^n_{old} \in \mathbb{N}$ captures the last round when n receives the old route $\mathcal{P}_{old}(n)$, and $r^n_{new} \in \mathbb{N}$ is the first round in which n receives $\mathcal{P}_{new}(n)$.

Next, we enforce BGP route propagation as described by the happens-before model in Section 5.2. A neighbor $m \in \mathcal{D}_{old}(n)$ must exist that selects its old route $\mathcal{P}_{old}(m)$ for longer than n selects $\mathcal{P}_{old}(n)$, i.e., $r^n_{old} < r^m_{old}$. Otherwise, n would change its routing decision uncontrollably after m does so. Symmetrically, another neighbor $m \in \mathcal{D}_{new}(n)$ must exist that selects its new route before n does, i.e., $r^n_{new} > r^m_{new}$. This yields

$$\bigwedge_{n \in N} r^n_{old} < \max_{m \in \mathcal{D}_{old}(n)} r^m_{old} \quad \wedge \quad \bigwedge_{n \in N} r^n_{new} > \min_{m \in \mathcal{D}_{new}(n)} r^m_{new}. \qquad (2)$$

When $r^n_{old} = r^n_{nh} = r^n_{new}$, then n knows its initial and its final route in the round when it is supposed to switch. On the contrary, if $r^n_{old} < r^n_{nh}$, then n must learn its initial route $\mathcal{P}_{old}(n)$ from somewhere else, so it can keep selecting its old route until round r^n_{nh}. Similarly, if $r^n_{nh} < r^n_{new}$, node n does not yet know its final route $\mathcal{P}_{new}(n)$ in the round when it should select it, so it must also learn it from somewhere else. In both cases, we must introduce a temporary BGP session. We minimize the number of such sessions as follows:

$$\min \sum_{n \in N} \left(1 \text{ if } r^n_{old} \neq r^n_{nh} \text{ else } 0\right) + \left(1 \text{ if } r^n_{nh} \neq r^n_{new} \text{ else } 0\right).$$

We encode all non-linear *if-then-else* operators with linear constraints using the big-M method [163]. We do the same for non-linear *min* and *max* operations, such as those in Eq. (2), which require that a variable is smaller than the largest one in a set of variables.

5.3.2 Concurrent Updates

Our ILP allows multiple nodes to update their next hop in the same round. As a result, individual updates of the same round may occur in any order. Hence, the ILP must guarantee the specification is satisfied in all possible orderings for each round. To that end, *Chameleon* ensures all forwarding state updates in the same round are *independent*. We define two forwarding state updates of node n_1 and n_2 to be independent if the forwarding path of n_1 before and after the update does not traverse n_2, and vice versa. Consequently, every forwarding path in the network experiences at most one change in each round

We recursively enforce the independence of all forwarding state updates in the same round. For each node $n \in N$ and round k, we introduce the boolean variable $\delta_k^n \in \mathbb{B}$ to capture whether either n or a different node along the forwarding path of n changes its next hop in round k. We express δ_k^n in terms of its old and new next hop $x = nh_{old}(n)$ and $y = nh_{new}(n)$:

$$\delta_k^n = \begin{cases} \delta_k^x & \text{if } r_{nh}^n > k \quad \text{\small n uses its old route in round k} \\ 1 + \delta_k^x + \delta_k^y & \text{if } r_{nh}^n = k \quad \text{\small n changes its route in round k} \\ \delta_k^y & \text{if } r_{nh}^n < k \quad \text{\small n uses its new route in round k} \end{cases} \tag{3}$$

In case $r_{nh}^n \neq k$, δ_k^n inherits its value from on either the initial or the final next hop. However, in case n changes its routing decision in round $r_{nh}^n = k$, then $\delta_k^n = 1+\delta_k^x+\delta_k^y$. Since $\delta_k^n \in \{0, 1\}$, this constraint implies that $\delta_k^x = 0$ and $\delta_k^y = 0$, i.e., the absence of other updates along n's old or new forwarding path.

5.3.3 *Specification*

To encode the specification ϕ in the ILP, *Chameleon* generates a *syntax tree* according to the specification language. Each node in that tree is labeled with its production rule, as shown in Figure 5.2. We then generate a Directed Acyclic Graph (DAG) $G_\phi = (\Phi, E_\phi)$ by combining equivalent nodes with identical descendants. In other words, we simplify the tree by combining branches that result in redundant constraints. In the following, we call each node $\phi_i \in \Phi$ in this syntax graph an *expression.* For each expression, we introduce symbolic boolean variables $\phi_{i,k} \in \mathbb{B}$, capturing whether ϕ_i is satisfied in round k. The following describes each kind of expression and their ILP constraints.

Our implementation ensures the uniqueness of this tree by surrounding each production rule with parenthesis.

Reachability. *Chameleon* uses recursive constraints to ensure reachability. Essentially, we say a node n is reachable $reach(n)$ if its next hop is so. This recursive requires checking the reachability on all nodes $m \in N$ in the network. More formally, for each node $m \in N$ and each round $k \in \{1, \dots, R\}$, we introduce a boolean variable $\phi_{reach,k}^m \in \mathbb{B}$:

$$\phi_{reach,k}^m = \begin{cases} 1 & \text{if } x = d \\ 0 & \text{if } x = \emptyset \\ \phi_{reach,k}^x & \text{otherwise} \end{cases}$$

where $x = nh_{new}(m)$ if $r_{nh}^m \leq k$ else $nh_{old}(m)$.

Here, nh_{old} and nh_{new} are the initial and final forwarding states, and x is the next hop of node m at round k. Notice that these constraints are sufficient if and only if there are no forwarding loops. We explain in Section 5.3.4 how we ensure the absence of loops.

Waypointing. Waypoints are treated similarly to reachability. Node n satisfies $wp(n, w)$ if its next hop does so or if its next hop is w. For each waypoint target $w \in N$ in the specification, we add a boolean variable $\phi^m_{wp(w),k}$ for each node $m \in N$ and round $k \in \{1, \dots, R\}$, constrained in terms of m's next hop x:

$$\phi^m_{wp(w),k} = \begin{cases} 1 & \text{if } x = w \lor n = w \\ 0 & \text{if } x \in \{\emptyset, d\} \\ \phi^x_{wp(w),k} & \text{otherwise,} \end{cases}$$
$$\text{where } x = nh_{new}(m) \text{ if } r^m_{nh} \leq k \text{ else } nh_{old}(m).$$

Logical and temporal modal operators. *Chameleon* encodes a logical or temporal modal expression ϕ_i by following its production rule and referring to the symbolic boolean variables of its sub-expressions. We constrain the variables $\phi_{i,k} \in \mathbb{B}$ as follows:

- We implement the conjunction rule $\phi_i = \phi_a \land \phi_b$ at round k using the big-M method [163]: $\phi_{i,k} = \phi_{a,k} \land \phi_{b,k}$. The other logical operators ($\neg$ and $\lor$) are constructed identically.
- We encode the *global* operator $\phi_i = \mathbf{G}\,\phi_a$ by asserting ϕ_a is satisfied indefinitely, starting from k: $\phi_{i,k} = \bigwedge_{j \geq k} \phi_{a,j}$.
- To encode the *until* operator $\phi_i = \phi_a\,\mathbf{U}\,\phi_b$, we assert the existence of a round $l \geq k$ until which ϕ_a is satisfied, and at which ϕ_b holds. To that end, we unroll the $\exists$ quantifier:

$$\bigvee_{l \geq k} \left(\phi_{b,l} \land \bigwedge_{m<l} \phi_{a,m} \right)$$

- We implement all other temporal modal operators similarly to $\mathbf{U}$ and $\mathbf{G}$ by unrolling logical quantifiers.

Ultimately, we assert the reconfiguration plan satisfies ϕ by asserting the root of the syntax tree ϕ_r holds in the first round $k = 1$, consistent with the semantics of LTL:

To assert any property ϕ is satisfied throughout the reconfiguration, users must specify $\mathbf{G}\,\phi$ instead.

$$\phi_{r,1} = 1.$$

5.3.4 Absence of Loops

The recursive constraints of reachability and waypoint require the absence of forwarding loops. To illustrate, let us consider a forwarding loop between n_1 and n_2. While generating the reachability constraints, *Chameleon* only requires $\phi^{n_1}_{reach} = \phi^{n_2}_{reach}$, thus allowing the model to choose an arbitrary value for $\phi^{n_1}_{reach}$.

We ensure the absence of forwarding loops by enumerating all possible forwarding loops. To that end, we build a graph $G_{nh} = (N, E_{nh})$ by combining the old and the new forwarding states. For each node $n \in N$, G_{nh} contains two edges from n to its old and new next hop, $nh_{old}(n)$ and $nh_{new}(n)$, i.e., $(n, nh_{old}(n)) \in E_{nh}$ and $(n, nh_{new}(n)) \in E_{nh}$. Each simple cycle in G_{nh} represents one possible forwarding loop. We enumerate all simple cycles L of G_{nh}. For each cycle $l = \langle n_1, \dots, n_j, n_1 \rangle \in L$ of length j and each round k, we assert that not all nodes n_i along the cycle can choose n_{i+1} as their next hop simultaneously:

$$j > \sum_{i \leq j} \begin{cases} 1 & \text{if } n_{i+1} = nh_{old}(n_i) \land r^{n_i}_{nh} > k \\ 1 & \text{if } n_{i+1} = nh_{new}(n_i) \land r^{n_i}_{nh} \leq k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The constraints on concurrent updates (cf. Section 5.3.2 and Eq. (3)) implicitly ensure the absence of loops if the initial forwarding state is loop-free. However, we notice the variance of the scheduling time of *Chameleon* decreasing when explicitly enumerating each possible loop, while the average time is unaffected. We, therefore, keep the explicit loop constraints in the model even though they are not strictly necessary.

Intuitively, Eq. (3) *already enforces the absence of forwarding loops in each round k, as δ^n_k counts the number of nodes along the forwarding path of node n in round k that also update their next hop in the same round. If there is a cycle that involves n, then there does not exist such a δ^n_k (as its value tends towards infinity).*

5.4 Compiler

Our compiler transforms a schedule of routing and forwarding state into a reconfiguration plan. This reconfiguration plan is a sequence of commands and conditions, forming synchronization barriers to implement the schedule. Each command will only affect a specific entry of a single router's routing and forwarding table. Similarly, each condition only needs to check a particular entry of the routing or forwarding tables. The following explains how the compiler transforms the 3-tuples $(r^n_{old}, r^n_{nh}, r^n_{new})$ from the scheduler into a reconfiguration plan.

To gradually transition the routing state as computed by the scheduler, *Chameleon* changes route attributes that stay local to that device without affecting other routers in the network. To that end, the compiled commands modify a route's weight [162] using route maps.

We could achieve the same using the local-preference, by increasing it in the incoming route map and decreasing it again in the outgoing one.

Reconfiguration phases. The reconfiguration plan consists of three phases: setup, update, and cleanup. In the update phase, we forbid *Chameleon* to switch between equivalent routes from different neighbors (cf. Section 5.2). Instead, we ensure each node appropriately changes its BGP neighbor in the setup phase. Based on the happens-before relations, there exists a neighbor $m^n_{old} \in \mathcal{D}_{old}(n)$ of node n that advertises $\mathcal{P}_{old}(n)$ to n for longer than r^n_{old}. Likewise, neighbor $m^n_{new} \in \mathcal{D}_{new}(n)$ advertises $\mathcal{P}_{new}(n)$ to n before round r^n_{new}. The setup phase ensures each node n prefers the route learned from m^n_{old}. Further, the setup phase establishes all temporary BGP sessions. Similarly, the cleanup phase removes all route preferences and temporary sessions.

We divide the update phase into R rounds. *Chameleon* ensures synchronization by *(i)* using pre- and post-conditions for commands and *(ii)* transitioning between rounds when all post-conditions are satisfied. Pre-conditions check a route's availability, while a post-condition ensures it is selected. We generate those commands and conditions using rules shown in Table 5.1 based on the 3-tuple $(r^n_{old}, r^n_{nh}, r^n_{new})$ of each node n. These rules guarantee node n selects $\mathcal{P}_{old}(n)$ until round r^n_{old}, updates its next hop in round r^n_{nh}, and then selects $\mathcal{P}_{new}(n)$ starting from round r^n_{new}.

Original reconfiguration commands. The reconfiguration plan must transition the network from the initial to the final configuration. *Chameleon* interleaves its temporary commands with the original reconfiguration commands. Let $N^* \subseteq N$ be all nodes subject to those commands. For each node $n \in N^*$, we apply its original commands c^* immediately before or after r^n_{nh}. If c^* makes n deny route $\mathcal{P}_{old}(n)$, then we execute c^* after r^n_{nh}. Otherwise, we schedule c^* before r^n_{nh}. Thus, we guarantee n knows $\mathcal{P}_{old}(n)$ until round r^n_{nh} and $\mathcal{P}_{new}(n)$ starting from r^n_{nh}.

Chameleon treats each destination separately. To support reconfigurations that affect multiple destinations, *Chameleon* performs the update phase for all destinations in parallel and aligns their execution along the original commands. For instance, assume that the original command c^* is applied before

Table 5.1: Compilation rules for the update phase.

$r^n_{old} = r^n_{nh} = r^n_{new}$: • In round r^n_{nh}, make n prefer $\mathcal{P}_{new}(n)$ from m^n_{new}. ◦ *pre-condition*:: n knows $\mathcal{P}_{new}(n)$. ◦ *post-condition*:: n selects $\mathcal{P}_{new}(n)$.
$r^n_{old} < r^n_{nh} = r^n_{new}$: • In round r^n_{old}, make n prefer the route from $e\big(\mathcal{P}_{old}(n)\big)$ using a temporary BGP session. ◦ *post-condition*:: n selects the route from $e\big(\mathcal{P}_{old}(n)\big)$. • In round r^n_{nh}, make n prefer $\mathcal{P}_{new}(n)$ from m^n_{new}. ◦ *pre-condition*:: n knows $\mathcal{P}_{new}(n)$. ◦ *post-condition*:: n selects $\mathcal{P}_{new}(n)$.
$r^n_{old} = r^n_{nh} < r^n_{new}$: • In round r^n_{nh}, make n prefer the route from $e\big(\mathcal{P}_{new}(n)\big)$ using a temporary BGP session. ◦ *post-condition*:: n selects the route from $e\big(\mathcal{P}_{new}(n)\big)$. • In round r^n_{new}, make n prefer $\mathcal{P}_{new}(n)$ from m^n_{new}. ◦ *pre-condition*:: n knows $\mathcal{P}_{new}(n)$. ◦ *post-condition*:: n selects $\mathcal{P}_{new}(n)$.
$r^n_{old} < r^n_{nh} < r^n_{new}$: • In round r^n_{old}, make n prefer the route from $e\big(\mathcal{P}_{old}(n)\big)$ using a temporary BGP session. ◦ *post-condition*:: n selects the route from $e\big(\mathcal{P}_{old}(n)\big)$. • In round r^n_{nh}, make n prefer the route from $e\big(\mathcal{P}_{new}(n)\big)$ using a temporary BGP session. ◦ *post-condition*:: n selects the route from $e\big(\mathcal{P}_{new}(n)\big)$. • In round r^n_{new}, make n prefer $\mathcal{P}_{new}(n)$ from m^n_{new}. ◦ *pre-condition*:: n knows $\mathcal{P}_{new}(n)$. ◦ *post-condition*:: n selects $\mathcal{P}_{new}(n)$.

round 3 for destination d_1 and after round 4 for d_2. We perform the update phase for d_1 until round 2 and d_2 until round 4. Then, we apply the c^* and proceed with the update phase of both d_1 and d_2. However, such an alignment may not exist if multiple nodes are subject to the original reconfiguration. In fact, the schedule for d_1 could require the original command c^*_1 to be applied after c^*_2, whereas d_2 expects the reverse order for c^*_1 and c^*_2. In that case, we use *Snowcap* to split the reconfiguration into commands that target individual nodes. We then apply each of them one by one in an order computed by *Snowcap*.

5.5 Case Study

We demonstrate the effectiveness of *Chameleon* in a testbed. The testbed consists of three Cisco Nexus 7000 devices, each running four virtual routers. We connect the resulting 12 virtual routers to a Barefoot Tofino programmable switch, enabling us to emulate any network with 12 routers. We simulate the physical distances between routers using a server that delays packets on all links. We provide more details in [9].

We use the above setup to emulate the *Abilene* network from Topology Zoo [115], a network with 11 nodes. We configure the routers to run OSPF [33] and connect them in an iBGP route reflection topology with three route reflectors. Then, we generate three external networks, each injecting routes towards 1024 individual prefixes at nodes e_1, e_2, and e_3. All nodes prefer routes from e_1 over e_2 and e_3 and decide between e_2 and e_3 based on the shortest IGP path. The reconfiguration scenario removes the BGP session between e_1 and its external peer, forcing all routers to change their routing decision. The network treats all prefixes identically; thus, *Chameleon* considers only a single prefix equivalence class.

During the reconfiguration, we require that all destinations remain reachable. In addition, each router must switch *exactly once* from its initial egress $e_1 = egress(\mathcal{P}_{old}(n))$ to its final one $e_n = egress(\mathcal{P}_{new}(n))$ (either $e_n = e_2$ or $e_n = e_3$). More precisely,

$$\phi = \bigwedge_{n \in N} \mathbf{G}\, reach(n) \wedge wp(n, e_1)\, \mathbf{U}\, \mathbf{G}\, wp(n, e_n). \tag{5}$$

To validate if *Chameleon* satisfies the specification, we inject traffic at a constant rate at each node towards d and measure the egress router where they leave the network.

Comparison with Snowcap. *Snowcap* applies the reconfiguration command directly to the network, as the reconfiguration affects only a single line in the configuration. Figure 5.1 shows the comparison between *Chameleon* and *Snowcap*. *Snowcap*'s reconfiguration takes 1.7 seconds to finish. However, during this time, the network drops around 15 k packets for almost one second of traffic. Further, around 1.3 k packets violate the waypoint constraints.

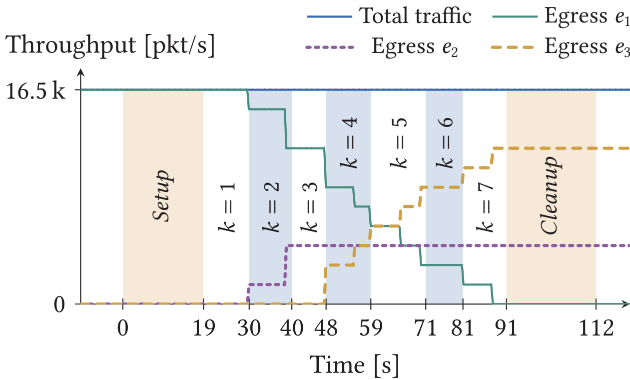


Figure 5.6: *Most rounds take 10–12 seconds to execute. The Shaded regions show the setup or cleanup phase and each round k in the update phase. Most time is spent waiting for the routers to update their route maps. Explore the reconfiguration plan interactively at bgpsim.github.io?s=abilene.*

In contrast, *Chameleon* applies the reconfiguration without dropping any packets and permanently preserves waypoint requirements. *Chameleon* reconfigures the network in around 112 seconds, of which 72 seconds are spent in the main update phase. Figure 5.6 shows *Chameleon*'s phases and rounds. We repeat the same experiment on five different topologies and configurations and show their results in Figure 5.7. All our experiments show a similar outcome: *Snowcap* always triggers transient black holes and usually violates waypoint constraints, while *Chameleon* performs all reconfigurations safely.

5.6 Evaluation

In this section, we evaluate the overhead of *Chameleon* in three dimensions. First, in Section 5.6.1, we analyze the *scheduling time* and show that it typically takes *Chameleon* a few minutes to schedule challenging BGP reconfigurations that affect the routing state of the entire network. We additionally capture the two major factors impacting *Chameleon*'s scheduling time with metrics that we call reconfiguration and specification complexity; we show that the former has a heavier impact than the latter. Second, in Section 5.6.2, we analyze the *reconfiguration time* and show that *Chameleon* can perform large reconfigurations in a few minutes while guaranteeing safety during the entire process. Third, in Section 5.6.3, we measure *routing table sizes* and compare *Chameleon*'s memory requirement with previous work: Snowcap does not need any extra memory, SITN roughly double, and *Chameleon* only around 10%.

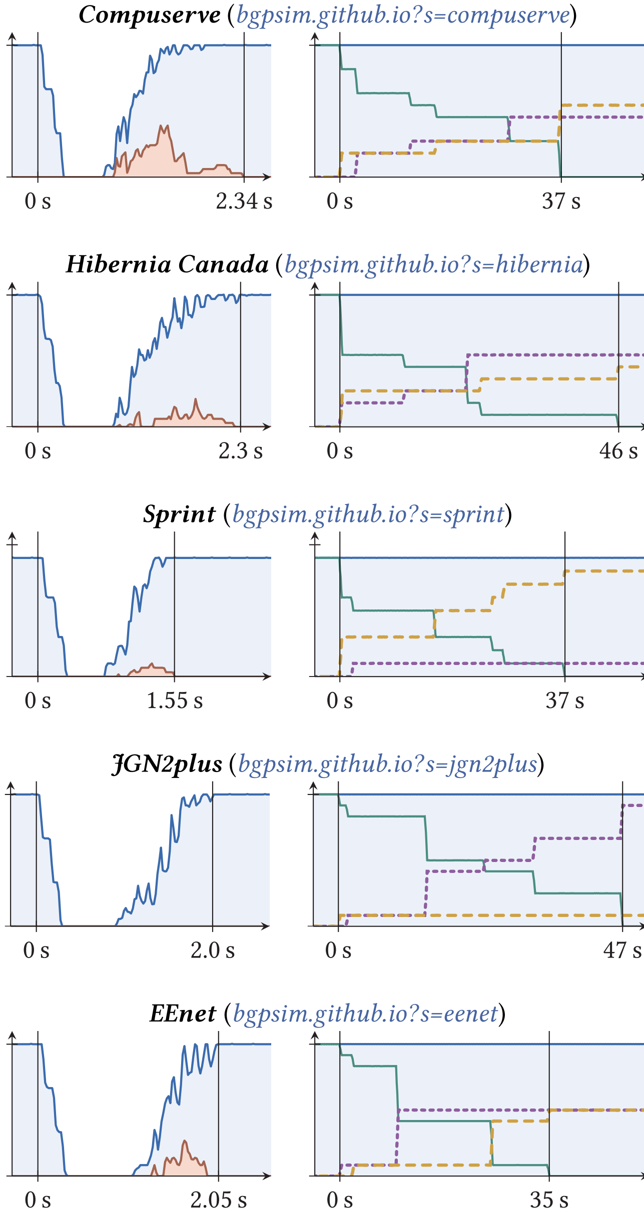


***Figure** 5.7: Chameleon (on the right) consistently and safely reconfigures a network, while Snowcap [5] (on the left) triggers black holes, and violates waypoint specifications. We repeat a similar experiment to Figure 5.1 on five additional topologies from Topology Zoo [115], all containing eleven or twelve routers. The blue line shows the total traffic routed through the network, while the red one visualizes packets violating the waypointing constraints.*

Topology	$\|N\|$	C_r	*scheduling time*
Deltacom	113	1810	23.4 s
Ion	125	2731	189.0 s
Pern	127	162	3.7 s
TataNld	145	3343	99.2 s
Colt	153	1194	15.6 s
UsCarrier	158	2505	50.5 s
Cogentco	197	4657	978.4 s

***Table** 5.2: The reconfiguration complexity C_r shows stronger correlation with scheduling time than the number of nodes in the network.*

Implementation. We implement *Chameleon* using ≈ 7 k lines of Rust code, including the analyzer, scheduler, compiler, and runtime controller, while supporting specifications as defined in Figure 5.2. We rely on our network simulator *BGPSim*, described in Section 2.4, to obtain the initial and final routing state and validate *Chameleon*'s reconfiguration plan for experiments that are too large to run directly on our testbed. To solve the ILP, we use COIN-OR CBC [164]. We run all experiments on a server with 32 CPU cores and 64 GB of memory. COIN-OR CBC is allowed to use all available cores.

The source code is available at github.com/nsg-ethz/chameleon (GPLv2 license)

Reconfiguration scenarios. We use 106 topologies from the Topology Zoo dataset [115]; their size ranges from a few routers to 754. For each topology, we randomly select three different nodes, e_1, e_2, and e_3, as border routers and connect them to three external networks. All external networks advertise the same destination d, and all internal nodes prefer routes received by e_1 over those from e_2 and e_3. The two routes from e_2 and e_3 have similar attributes, so routers pick their preferred one based on their distance in IGP to e_2 and e_3. We then elect three random nodes to be BGP route reflectors; every other router is a client of all three reflectors.

We design reconfiguration scenarios to affect all routers and their routing state to evaluate *Chameleon* in a challenging setting. Adding or removing BGP sessions are reported to be the most common BGP reconfigurations [25], but they rarely affect all routers in the network. Instead, we simulate adding a route map to the initially most preferred egress point e_1: the route map denies the route towards d from its external peer, forcing all routers in the network to change their selected route during the reconfiguration.

5.6.1 Scheduling Time

We first evaluate the scheduling time of *Chameleon*. Perhaps surprisingly, the time spent solving the ILP model is not directly proportional to topological metrics such as the network size. Table 5.2 shows the ILP-solving time for different networks and compares them with the number of nodes.

We investigate the most important factors that impact the ILP solving time. We identify two orthogonal dimensions: the reconfiguration complexity and the specification complexity. Intuitively, the reconfiguration complexity measures how many nodes each router can "reach" during a reconfiguration, which is agnostic to the invariants to be preserved, i.e., the specification. In other words, it relates to how many nodes could be traversed in any forwarding path during the reconfiguration. In contrast, the specification complexity measures how many variables are needed to track the specification's invariants, irrespective of the network topology, routing, and forwarding.

In the following, we separately evaluate how *Chameleon*'s scheduling time varies with respect to reconfiguration and specification complexity. To do so, we evaluate the impact of reconfiguration complexity by keeping the same specification and simulating different reconfigurations across all networks in our dataset, shown in Figure 5.8. We then assess the impact of the specification complexity by using specifications of increasing complexity for the same network and reconfiguration scenario, shown in Figure 5.9.

Reconfiguration complexity. Some reconfigurations affect a few nodes, and some affect nodes located far away from each other. Other scenarios, however, can update multiple nodes along a single forwarding path, thus introducing dependencies. We can schedule one scenario in a few milliseconds while spending minutes on a different scenario for the same network.

We measure the number of dependencies between nodes by defining the reconfiguration complexity C_r. Intuitively, C_r counts, for each node n, the number of different nodes that forwarding paths of n could traverse during the reconfiguration. Yet, we only count those nodes along a path that also update their next hop, as this inherently links the two next-hop updates in that they cannot happen in the same round.

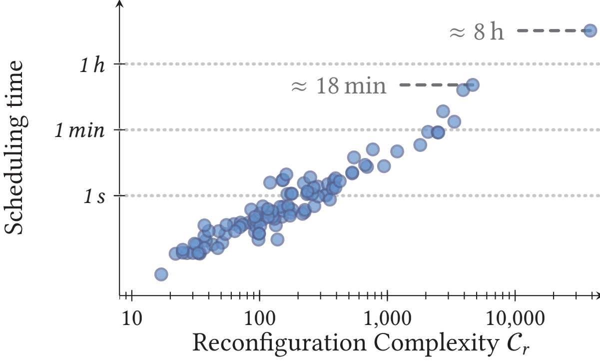


Figure 5.8: *This figure shows a strong correlation between the reconfiguration complexity C_r and the scheduling time. Chameleon usually finds a schedule in a few minutes and up to 8 hours for the largest scenario, in which 660 routers change their next hop. Each point shows one of 106 scenarios, and both the x- and y-axis have a logarithmic scale.*

To find C_r, we construct a directed graph $G_{nh} = (N, E_{nh})$ as the union of the initial and final forwarding state nh_{old} and nh_{new}, as described in Section 5.3.4. Further, we define the set $N_{nh} = \{n \in N \mid nh_{old}(n) \neq nh_{new}(n)\}$ be the set of all nodes that eventually change their next hop, and let $reachable(G_{nh}, n)$ contain all reachable nodes in G_{nh} from n. C_r is defined as:

E_{nh} contains edges $(n, nh_{old}(n))$ and $(n, nh_{new}(n))$ for each node $n \in N$.

$$C_r = \sum_{n \in N_{nh}} \left| reachable(G_{nh}, n) \cap N_{nh} \right|.$$

Note that C_r only depends on the initial and final forwarding states nh_{old} and nh_{new} but does not on the specification ϕ.

Figure 5.8 displays a log-log plot that shows the strong correlation between the reconfiguration complexity C_r and the scheduling time of *Chameleon*. Each point represents one reconfiguration scenario. *Chameleon* finds a solution in a few minutes despite the non-trivial specification ϕ from Eq. (5) and the network-wide reconfiguration scenario. In the worst case, it takes our system less than 8 hours to schedule a reconfiguration in which 754 nodes update their routing state, and 660 nodes change their next hop. Yet, for the vast majority of scenarios, *Chameleon* finds a schedule in less than a minute.

We explain the correlation between C_r and the scheduling time based on the semantics of our ILP constraints. Indeed, those related to the specification and to concurrent updates are recursive: a variable x_k^n associated with node n at round k depends on the next hop of n at round k, see Section 5.3. Consequently, x_k^n depends on the next hop of all nodes along all possible forwarding paths of n that also change their next hop, which C_r precisely captures.

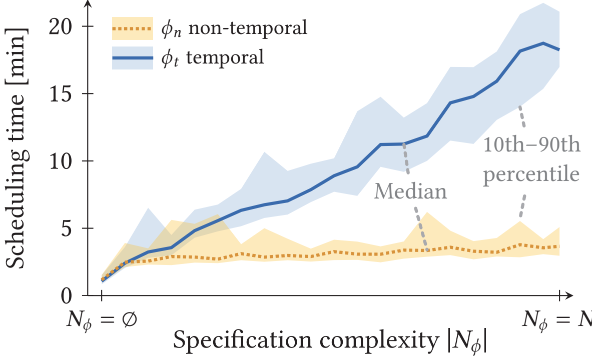


Figure 5.9: *Temporal expressions have a more significant impact on Chameleon's scheduling time than non-temporal constraints. This plot shows the scheduling time for the same scenario but with different specification complexities. When $N_\phi = \emptyset$, the used specification requires network-wide reachability, while the specification at $N_\phi = N$ contains temporal (in blue) or non-temporal (in orange) waypoint constraints for all nodes.*

Specification complexity. The specification language shown in Figure 5.2 captures specifications ranging from reachability to complex waypoint constraints that change over time. We evaluate how the complexity of the specification influences the scheduling time to determine the impact of *(i)* recursive properties like waypoints and *(ii)* temporal operators.

To this end, we define two different specifications: ϕ_t does contain *temporal* operators in the form **U G**, and ϕ_n asserts the same properties throughout the reconfiguration. We construct ϕ_t and ϕ_n as follows: Both specifications require reachability for all nodes and waypoint constraints for *some* nodes N_ϕ. Increasing the number of nodes in N_ϕ influences the complexity of the specification. Each node $n \in N_\phi$ must use either its initial egress router $e_1 = egress(\mathcal{P}_{old}(n))$ or its final one $e_n = egress(\mathcal{P}_{old}(n))$. In addition, the temporal specification ϕ_t further requires that node $n \in N_\phi$ switches *once* from e_1 to e_n. More precisely,

$$\phi_n = \bigwedge_{n \in N} \mathbf{G}\, reach(n) \wedge \bigwedge_{n \in N_\phi} \mathbf{G}\left(wp(n, e_1) \vee wp(n, e_n)\right)$$

$$\phi_t = \bigwedge_{n \in N} \mathbf{G}\, reach(n) \wedge \bigwedge_{n \in N_\phi} wp(n, e_1)\, \mathbf{U}\, \mathbf{G}\, wp(n, e_n)$$

Figure 5.9 shows the scheduling time for a varying number of nodes in N_ϕ. The figure refers to reconfigurations on the *CogentCo* topology, the network with 197 nodes for which both C_r and the scheduling time are the second largest of all scenarios (i.e., the scenario with a scheduling time of 18 minutes in Figure 5.8). We run every experiment 20 times for each N_ϕ and plot the median scheduling time, as well as the 10th and 90th percentiles for both ϕ_n in orange and ϕ_t in blue.

The figure highlights a significant difference between the non-temporal specification ϕ_n and the temporal one ϕ_t: While adding more waypoints has a small impact on *Chameleon*'s scheduling time, adding temporal operators has a significant effect. This disparity is because waypoint constraints are by nature recursive; constraints for a single waypoint target are added for all nodes, even if only one node must use that waypoint target. In contrast, the temporal operators in ϕ_t demand variables and constraints for each additional node in N_ϕ.

Takeaways. *Chameleon*'s scheduling time depends on the reconfiguration and specification complexity, two orthogonal dimensions. Yet, the scheduling time is much more sensitive to the reconfiguration complexity. In fact, the specification complexity increases the scheduling time by 20×, while the reconfiguration complexity does so by four orders of magnitude.

5.6.2 Reconfiguration Time

We show that *Chameleon* applies large-scale reconfigurations in a few minutes while guaranteeing the input specification ϕ to be satisfied in all transient states. Our testbed includes only 12 virtual routers. To estimate the reconfiguration time of *Chameleon* in larger networks, we run simulations.

The time for *Chameleon* to apply a single round depends on two factors: the time to apply the reconfiguration and the convergence time itself. From our measurements in Figure 5.6, we determine that routers in our testbed take between 8 and 12 seconds to modify BGP route maps (presumably to anticipate additional changes, such that updates can be applied in batches). In contrast, the convergence time is negligible. This is because *Chameleon* enforces the happens-before relations: updating the selected route of one node will never change the selected route of a different node. Consequently, any BGP update will be ignored, and the convergence time depends on the distance between nodes rather than on the network size.

We find that newer hardware like the Cisco Nexus 9000 series applies such updates instantly.

We approximate the running time $\tilde{t}$ of a reconfiguration plan by counting its number of rounds R, including the setup and cleanup phase. To that end, we define $\tilde{t}_{rm}$ as the approximation of the time to apply a single round of the reconfiguration plan. We approximate the reconfiguration time as

$$\tilde{t} = \tilde{t}_{rm}\,(2 + R)\,.$$

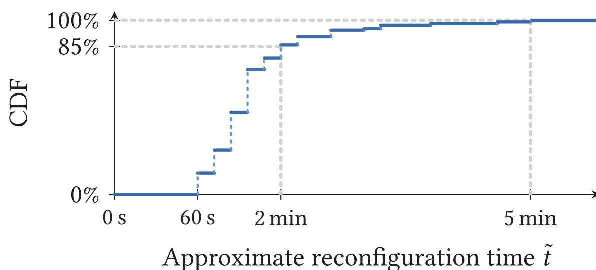


Figure 5.10: *The Cumulative Distribution Function (CDF) of the reconfiguration time $\tilde{t}$ for all 106 scenarios. We approximate Chameleon to reconfigure 85% of scenarios in less than two minutes.*

The reconfiguration time heavily depends on the specific hardware. Our case study suggests $\tilde{t}_{rm} \approx 12\,\text{s}$ for Cisco Nexus 7000 routers. In the following, we use $\tilde{t}_{rm} = 12\,\text{s}$.

Further, we measure no significant impact of routing table size on $\tilde{t}_{rm}$, as we have compared $\tilde{t}_{rm}$ for table sizes between 1 and 16 k.

Results. We estimate the running time for all 106 scenarios and find that *Chameleon*'s approximate reconfiguration time is less than 2 minutes for 85% of the scenarios. For the largest reconfiguration scenario, we estimate that *Chameleon* takes 5 minutes to reconfigure all 754 nodes. We provide a cumulative distribution function of the approximated reconfiguration time across all 106 scenarios in Figure 5.10.

5.6.3 Routing Table Size

In the following, we compare *Chameleon* with related work regarding additional routing table entries needed during the reconfiguration process. On average, *Chameleon* replicates about 11% of the routing state to preserve invariants during the transient state, whereas related work duplicates all routing processes to achieve similar guarantees. For *Chameleon*, the routing table size increases depending directly on the number of temporary BGP sessions—the scheduler can minimize the required number of temporary sessions to account for networks or routers with specific memory limitations.

Methodology. While simulating *Chameleon* to reconfigure a network, we measure the maximum number of routing table entries in the network at any given time during the process. We then compare this distributed routing table size of *Chameleon* with the baseline of directly reconfiguring the network (as Snowcap would also do) and SITN [25].

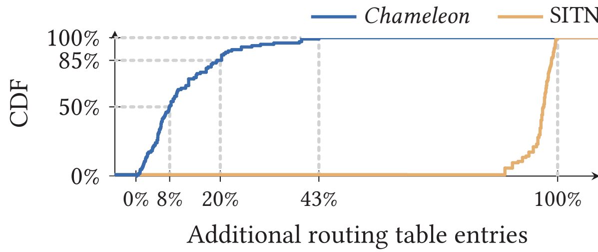


Figure 5.11: *In the median case, Chameleon requires 8% additional routing table entries compared with the baseline, while SITN requires 96%. This figure shows the CDF of the additional routing table entries required by Chameleon and SITN, normalized by the maximum table size used for the baseline.*

Results. *Chameleon* requires, on average, 11% more routing table entries than the baseline, whereas SITN requires 96% more entries. Figure 5.11 shows detailed statistics for all 106 scenarios. We note that temporary BGP sessions exclusively cause state replication in *Chameleon*'s reconfigurations. Hence, *Chameleon* can customize its reconfiguration plans according to specific memory limitations of operational networks. For example, if a given node n is operating at its memory's limit, *Chameleon* can generate a reconfiguration plan in which n only uses existing BGP sessions by including a constraint $r^n_{old} = r^n_{nh} = r^n_{new}$.

5.7 Limitations and Discussion

Specifications. We designed *Chameleon* to guarantee that, for each BGP destination, every router always selects its best BGP route in the initial configuration or the final one. We choose to avoid transiently leaking intermediate routes to neighboring networks, preventing unnecessary interdomain churn.

The drawback of this choice is that *Chameleon* may not be able to compute a safe reconfiguration for highly constrained specifications. While *Chameleon* can *always* guarantee reachability, it may be unable to carry out a safe reconfiguration in the presence of many waypointing invariants. This is because *no safe reconfiguration actually exists* in some settings if routers are only allowed to use their initial and final BGP routes. In those cases, *Chameleon* notifies the user that it cannot perform the reconfiguration safely.

We expect that, in practice, *Chameleon* can safely perform most reconfigurations, while failing to do so very rarely. In all our experiments in Section 5.6, we only encountered two examples of unsolvable specifications: they try to enforce waypointing requirements in a star-shaped topology.

We plan to extend *Chameleon* so that operators can trade consistency of routes announced to neighboring networks for the feasibility of highly constrained reconfigurations. This necessitates explicitly supporting *routing invariants*, that is, constraints on the BGP routes selected by routers during the reconfiguration. Supporting such routing invariants should be straightforward with *Chameleon*'s design: in addition to updating the specification language (see Figure 5.2), it would involve formulating new happens-before relations for routing invariants and translating them into ILP constraints.

Dependencies between prefixes. *Chameleon* treats all prefixes independently. If a reconfiguration impacts multiple prefixes, *Chameleon* computes the reconfiguration plans for each prefix while solving the resulting smaller problems in parallel and performing per-prefix reconfiguration concurrently. The above approach is consistent whenever eBGP routes are not modified in iBGP, as is often suggested by current best practices [37].

More advanced BGP configurations can, however, create dependencies between prefixes, such as route aggregation/de-aggregation or iBGP policies.

- Most BGP speakers support *route aggregation*, where nodes aggregate multiple routes into a single one and advertise the summary to neighboring routers. We note that only routes aggregated *within* a network create dependencies between prefixes. Typically, routes are aggregated at the network border to either reduce the number of routes handled in iBGP or to announce a single eBGP route to external networks [165]. In those common cases, it is still correct to treat prefixes independently since internal routers know either all individual routes or only the summarized ones. Similar considerations hold when prefixes are de-aggregated in sub-prefixes, for example, for traffic engineering [166].
- Routers may apply arbitrary *iBGP policies* using route maps, i.e., modifications of BGP routes when crossing internal BGP sessions. Some Internet networks do configure iBGP policies [167]. Depending on the configured route maps, this practice can create dependencies between prefixes. For example, if route maps discard a route on internal BGP sessions, different routers may forward the same packet by matching it to more or less specific prefixes, depending on which routes they receive.

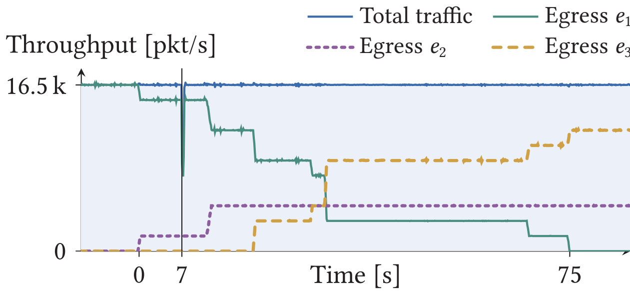


Figure 5.12: *A link failure during the reconfiguration at 7 s causes OSPF to reconverge. Some packets are lost for ≈ 0.5 s, but Chameleon causes no additional disruption.*

Chameleon's scheduler can be extended to support these cases by jointly considering multiple prefixes. Supporting route aggregation requires generalizing our current happens-before relations. Further, modeling the absence of routes at specific nodes can be encoded in the next hop computation of the ILP.

External events. *Chameleon* guarantees correctness during the reconfiguration process in the absence of external events, such as link failures or external route changes. Stated differently, *Chameleon* might (transiently) violate some of the invariants if external events happen during the reconfiguration.

Because of the commands applied by *Chameleon*, *only the withdrawals of the best BGP routes* can break our correctness guarantees. By definition, no routers select non-best routes, making those routes irrelevant. New best routes that appear during the reconfiguration are also irrelevant since transient states installed by *Chameleon* make routers ignore the new routes until the end of the reconfiguration process. In contrast, Snowcap is vulnerable to all kinds of BGP events.

As an illustration, we simulate unexpected external events while *Chameleon* reconfigures our testbed (see Section 5.5). Figure 5.12 shows the effect of a link failure that causes OSPF to reconverge during the reconfiguration. OSPF takes around half a second to reconverge, causing routers to drop packets during that time. The effect of the link failure would have been similar if it had occurred before or after the reconfiguration.

Figure 5.13 shows that the network reacts to the appearance of a new, more preferred route advertised to e_4 after *Chameleon* restores the original route preferences. That is, routers choose a sub-optimal path for about 70 seconds, but safety is preserved throughout the reconfiguration.

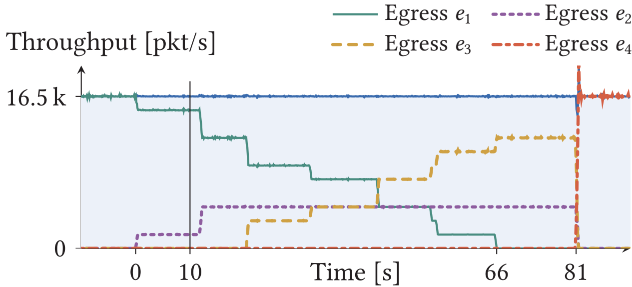


Figure 5.13: *A new route that is announced at 10s is ignored during the update phase. At 81 s, when Chameleon restores the original preferences, all nodes converge to the new route from e_4.*

We envision that *Chameleon*'s runtime controller can track the network state and external events, especially BGP withdrawals, and react to them as follows:

1. If the event does not impact correctness, we can delay its effect by simply ignoring it, similar to Figure 5.6.
2. If the event causes a suboptimal routing state, we can quickly compute a new reconfiguration plan to reduce the time spent in a suboptimal routing state. This is enabled by *Chameleon*'s scheduling efficiency, as we demonstrate in Section 5.6.1.
3. If the event triggers long-term anomalies, *Chameleon* should immediately commit to the final configuration so that it can restore connectivity as quickly as possible.

5.8 Related Work

An extended volume of proceeding work from academia and industry has studied the general network update problem [96]. While some operate on traditional (distributed) networks, the majority of systems [94, 100, 101] extend SDN, in which a central controller modifies the forwarding tables of all nodes. They can provide strong guarantees like per packet consistency by leveraging direct control of the forwarding decision.

As for traditional network update systems, we identify two broad categories. The first category of tools [93, 104] duplicates the control and data planes on all routers, allowing them to run both the initial and the final configurations in parallel. They then gradually instruct routers to forward traffic according to the final configuration in an order that avoids forwarding loops [24, 25]. While useful, these tools tend not to be used in practice due to their overhead.

The second category of tools [5, 27, 168] reconfigures a network "in-place" by partitioning the configuration changes into smaller units and gradually applying them to the network while preserving correctness. Compared to the first category, "in-place" reconfiguration imposes virtually no overhead, nor does it require router support. Prior techniques like [27, 168] can avoid forwarding loops during IGP reconfigurations but are not applicable to BGP. Snowcap [5] is the only in-place system to reconfigure BGP. However, it provides correctness guarantees only in steady states.

The literature has proposed many models for distributed routing protocols, particularly BGP, to verify properties of the network's steady state [38, 108, 135]. Most notably, the Stable Paths Problem [107] formulates conditions for a network to converge to a unique state. However, they cannot describe *how* the network converges. In contrast, *Chameleon* models transient states during convergence.

Finally, research has proposed extensions to BGP [169] to guarantee the absence of forwarding loops during convergence. Unfortunately, they have seen little adoption due to additional control logic introduced to BGP speakers.

5.9 Conclusion

Chameleon is the first system to apply BGP reconfiguration while maintaining correctness throughout the entire reconfiguration process. It is built around a framework to describe the convergence process of BGP and introduces techniques to generate a reconfiguration plan that seamlessly transitions the network from the old to the new configuration. *Chameleon* schedules and performs large reconfigurations in real networks within a few minutes while satisfying complex specifications.

We pose the BGP reconfiguration problem as a synthesis problem: find a sequence of configuration commands that safely transition the network from the old to the new configuration. In the next chapter, we focus on the generalized synthesis problem: find a configuration that satisfies a given specification.

6 On the Complexity of Network-Wide Configuration Synthesis

Configuring IP networks is a complex task nowadays. Network operators design low-level configurations of possibly hundreds of devices running complex distributed routing protocols to meet high-level requirements on packet forwarding, such as reachability, isolation, or load-balancing. Configuring networks this way is error-prone, often leading to downtimes [11, 12, 15]. In fact, Alibaba recently revealed that the majority of their network outages resulted from configuration mistakes [160].

Configuration synthesis aims to prevent such network outages by *automatically* generating low-level configurations out of a set of high-level forwarding properties. Using synthesis tools, operators only need to specify *what* should happen to their network traffic, as opposed to *how*.

A key challenge of configuration synthesis is scalability. While some systems use domain-specific knowledge to achieve better scalability for specific network topologies, e.g., how *Propane/AT* [87] leverages symmetries in data centers, other tools like *Zeppelin* [30] rely on a best-effort approach without guarantees for the existence or absence of a solution. Current generic synthesis tools, however, cannot support topologies larger than a few tens of routers [29, 85]. Perhaps frustratingly, though, it is especially in large networks that operators would benefit the most from synthesis tools.

Generally speaking, the complexity of synthesizing correct configurations comes from two ingredients:

(i) the forwarding properties that need to be implemented;
—the space of inputs of the synthesis problem—

(ii) the routing protocol(s) we synthesize configurations for.
—the space of outputs of the synthesis problem—

It is a priori not clear how these two dimensions interact. Are some protocols are *easy* to synthesize configurations for? Are some forwarding properties *hard* to satisfy independently of the used protocols?

Specific synthesis problems have been shown to be *NP*-hard [30, 170]. Despite that, these questions have not been comprehensively answered by the literature thus far.

In this chapter, we systematically analyze the computational complexity of network-wide configuration synthesis by exploring the problem space spanned by forwarding properties and routing protocols. We answer the questions stated above affirmatively: synthesizing configurations that enforce some forwarding properties is fundamentally hard. At the same time, synthesizing configurations for some protocols is easy in the case of practical forwarding properties.

We capture the space of routing protocols by an abstract *routing algorithm*, that is, any combination of destination-based hop-by-hop routing protocols. We identify five generic families of routing algorithms ranging from Shortest Path Routing (SPR) protocols to *non-uniform* algorithms, i.e., protocols that allow a separate routing configuration for each destination prefix. While synthesizing bounded integer link weights for SPR (e.g., OSPF) is known to be *NP*-hard [170], we show that non-uniform algorithms, such as routing exclusively with the BGP, allow for efficient configuration synthesis.

We exclude source routing, like MPLS, and segment routing, such as SRv6. Configuring such protocols involves explicitly specifying each path directly.

Regarding forwarding properties, we identify a class of fundamental path properties ranging from specifying entire paths to specifying waypoint constraints that mandate packets to cross a specified device. We show that, perhaps surprisingly, determining the satisfiability of a set of arbitrary waypoint constraints is already *NP*-hard for any hop-by-hop routing algorithm that can be configured to implement an arbitrary forwarding spanning tree such as OSPF, IS-IS, or BGP, see Theorem 6.4. While link weights for SPR protocols can be found by inverting the shortest-path computation for a set of fully specified paths, the problem becomes *NP*-hard as soon as some path segments are unspecified.

In summary, the main contributions of this chapter are:

- a taxonomy of the computational complexity (Table 6.1) of network-wide configuration synthesis regarding different forwarding properties and routing algorithms;
- a set of proofs showcasing that configuration synthesis for common routing algorithms is *NP*-hard;
- the identification of a family of routing algorithms and a subset of forwarding properties that can enable scalable network-wide configuration synthesis.

Routing Algorithm $\mathcal{A}$ / Specification $\mathcal{S}$ (forwarding properties)		$\mathcal{S} = \mathcal{P}$ Paths only	connected $\mathcal{S} = \mathcal{P} \cup \mathcal{V} \cup E$ Paths and connected waypoints	$\mathcal{S} = \mathcal{P} \cup \mathcal{V} \cup E$ Paths and waypoints
Shortest Path Routing with finite bit representation	OSPF or IS-IS	*NP*-Complete [170]	⇒ *NP*-Complete	⇒ *NP*-Complete
Shortest Path Routing (Section 6.2.1)	–	*P* $\mathcal{O}((n^2 + e)^{2.5})$ [171]	*NP*-Complete Th. 6.1	⇒ *NP*-Complete
Routing with separate propagation (Section 6.2.2)	BGP + RR + SPR	*NP*-Hard Th. 6.2	⇒ *NP*-Hard	⇒ *NP*-Hard
Hierarchical routing (Section 6.2.3)	Hierarchical BGP + SPR	*NP*-Hard Th. 6.3	⇒ *NP*-Hard	⇒ *NP*-Hard
Non-uniform routing (Section 6.2.4)	BGP exclusively	*P* $\mathcal{O}(d \cdot e)$ ⇐	*P* $\mathcal{O}(d \cdot e)$ Th. 6.5	*NP*-Hard Th. 6.4

***Table 6.1**: The computational complexity of configuration synthesis* Synth$(\mathcal{A}, \mathcal{S})$ *depends on both the target routing algorithm $\mathcal{A}$ (rows) and the forwarding properties $\mathcal{S}$ (columns, cf. Figure 6.1). For those problems solvable in polynomial time, its asymptotic complexity is given as a function of the number of routers n, links (edges) e, and destination prefixes d.*

6.1 Model

We define the problem of network-wide configuration synthesis Synth$(\mathcal{A}, \mathcal{S})$ as a family of algorithmic problems, one for each combination of an abstract routing algorithm $\mathcal{A}$, that is, an abstract representation of the deployed routing protocol(s) and a set of supported forwarding properties $\mathcal{S}$. An instance of Synth$(\mathcal{A}, \mathcal{S})$ takes as inputs:

- a *network topology* G;
- a set of *routing inputs*, i.e., a set of destination prefixes D, for both internal and externally advertised prefixes, and where each of them is advertised;
- a *network specification* ψ, that is, a set of the supported forwarding properties that the network must satisfy in its converged state;

and produces as an output a set of parameters for the selected routing algorithm $\mathcal{A}$, for example, link weights or route maps. To analyze the computational complexity of Synth$(\mathcal{A}, \mathcal{S})$, we consider the corresponding decision problem, that is, whether or not the selected routing algorithm $\mathcal{A}$ can implement a given subset of the supported forwarding properties, cf. Figure 6.1.

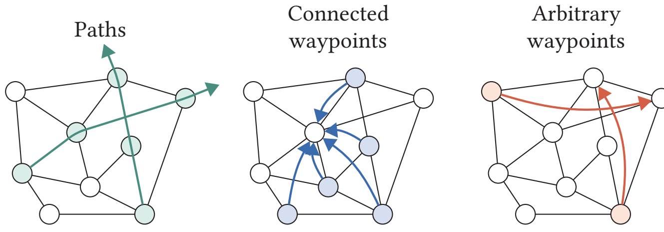


Figure 6.1: The type of forwarding properties influence the complexity of SYNTH$(\mathcal{A}, \mathcal{S})$ *as they vary in degree of freedom, i.e., the number of valid forwarding graphs.*

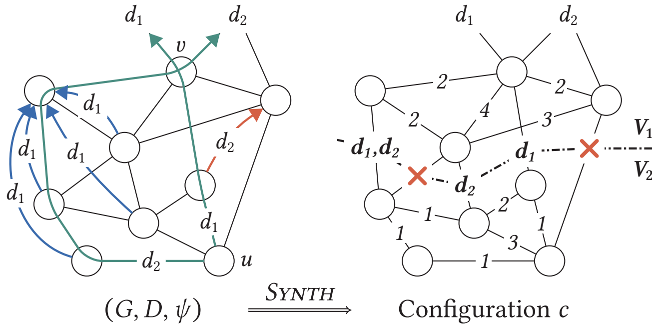


Figure 6.2: The synthesis problem is finding network configurations to satisfy a specification ψ. The left side shows a problem instance of SYNTH$(\mathcal{A}, \mathcal{S})$ *with two destinations $D = \{d_1, d_2\}$. The configuration for OSPF and BGP Confederations on the right solves this problem with two partitions V_1 and V_2. Edge labels inside partitions are OSPF weights, while those on partition boundaries indicate which destinations are allowed on that edge.*

To illustrate an instance of *SYNTH*$(\mathcal{A}, \mathcal{S})$, consider Figure 6.2. On the left, we depict a network topology with two destinations d_1 and d_2 as routing inputs, and the specification that consists of two path properties in green, a set of connected waypoints in blue, and a single arbitrary (disconnected) waypoint in red. Notice that the green path properties from u to d_1 and d_2 require two different sub-paths from u to v. This problem has, therefore, no solutions for SPR protocols such as OSPF that treat all destinations identically, i.e., for uniform routing algorithms.

Nevertheless, some routing algorithms *can* be configured to satisfy the specification. The right side of Figure 6.2 shows such a configuration. It splits the network into two partitions using BGP Confederations [172] and computes shortest paths with OSPF within each partition. BGP Confederations permit route maps between two partitions, such as V_1 and V_2, to filter routes based on the destination prefix. By only allowing routes for d_1 to propagate on the direct path, packets from u to d_2 take the desired detour.

In the following, we describe the network model and the two dimensions of *SYNTH*$(\mathcal{A}, \mathcal{S})$, namely the different types of network specifications and routing algorithms we consider.

6.1.1 Network Model

We encode the network topology as a directed (multi-)graph $G = (V, E)$, where we denote routers by the set of nodes V and links by directed edges E. Each edge $e = (u, v)$ has a source $src(e) = u$ and a destination $dst(e) = v$. A forwarding path $p = \langle e_n, \dots, e_1 \rangle$ is a sequence of contiguous and loop-free edges with source $src(p) = src(e_n)$ and destination $dst(p) = dst(e_1)$. We write the concatenation of two paths p and q as $p \circ q$ and denote $p_{k,i} = \langle e_k, e_{k-1} \dots, e_i \rangle$ to be a sub-path of p. Finally, $\mathcal{P}$ denotes the set of all paths in G.

We intentionally reverse the path indices to emphasize that routing information typically flows in the opposite direction of the forwarding path. Routing information first traverses edge e_1.

6.1.2 Forwarding Properties

The network specification can be understood as an invariant on the forwarding paths computed by a distributed routing algorithm, i.e., the converged forwarding state. Operators need to translate their intent on how the network should behave into invariants on that forwarding state—we call these *forwarding properties*. Ideally, operators could use high-level invariants, e.g., asserting that all nodes prefer one neighboring AS over another [38] or all traffic from a critical domain to first traverse a firewall. However, the space of invariants on the forwarding behavior is huge and the support for complex invariants has been very limited so far [29, 30].

Instead, we focus our analysis on a class of fundamental forwarding properties that constrain forwarding paths. These path constraints can be combined in various ways to express the forwarding behavior of an entire network.

We can restrict forwarding paths in many ways, ranging from specifying a complete path from a source node s to a destination d, which we call a *path property*, to being agnostic of the path as long as packets eventually reach their destination, i.e., *reachability*, or pass a specified device, i.e., a *waypoint*. Since most routing protocols provide reachability by default, synthesizing reachable configurations is often trivial. Hence, we limit our analysis to:

For instance, configuring OSPF to achieve reachability necessitates enabling OSPF on all routers and advertising all connected networks.

- *path* properties that fix the entire forwarding path, and
- *waypoint* properties that require a specific node or edge to be traversed by the forwarding path.

Formally, we model a specification as a partial function $\psi : (V \times D) \rightharpoonup \mathcal{S}$, where D is the set of destination prefixes. With slight abuse of notation, let

Note that this restricts each node to a single forwarding property per destination. Expressing multiple properties for a node and destination pair leads to inconsistencies.

$$\psi(s, d) = \begin{cases} p \in \mathcal{P} & \text{for } \textit{path} \text{ properties,} \\ \omega \in (V \cup E) & \text{for } \textit{waypoint} \text{ properties.} \end{cases}$$

We write $p_{s,d} \models \psi(s, d)$ to indicate a path $p_{s,d}$ from s to d satisfies the property $\psi(s, d)$. By the nature of hop-by-hop routing, we assume that any path property $\psi(s, d) = \langle e_n, \dots, e_1 \rangle$ implies the subsidiary path property $\psi(dst(e_n), d) = \langle e_{n-1}, \dots, e_1 \rangle$.

A forwarding path $\langle e_n, e_{n-1}, \dots, e_1 \rangle$ in any hop-by-hop routing protocol implies that the first hop $dst(e_n)$ forwards traffic along $\langle e_{n-1}, \dots, e_1 \rangle$.

Connected Waypoints

In practice, operators rarely need to enforce arbitrarily complex waypoints. Instead, they typically translate a single higher-level invariant into a set of waypoint properties that relate to each other. For instance, recall the examples from the beginning of this section: modeling AS preferences or enforcing the traversal of a firewall. Both of these can be expressed as a structured set of waypoint constraints by adding the same waypoint to an entire partition of the network. We observe that *(i)* the shared waypoint is located at the edge of this partition and that *(ii)* the subgraph induced by the nodes with this waypoint constraint is *connected*. Hence, it is guaranteed that there exists a forwarding tree rooted in ω that satisfies all of these waypoint constraints. More formally:

Definition 6.1. *A specification ψ is* connected *if and only if, for any waypoint property $\psi(s, d) = \omega$, there exists a path $p \in P$ from s to ω such that, for every node $n \in p$, there exists a property $\psi(n, d) = x$ that implies ω. In other words, any such path from n to d that satisfies $p_{n,d} \models x$ must satisfy $p_{n,d} \models \omega$.*

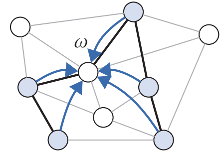


Illustration of connected waypoints and a possible forwarding tree rooted at ω.

Finally, observe that any computational complexity result on implementing these basic forwarding properties extends to many more complex network properties. To illustrate this, consider trying to find a configuration that satisfies path preferences $p_a > p_b$ under failure scenarios: A node should prefer the active path p_a as long as it is available and switch to a backup path p_b only if any link in p_a fails. We claim that synthesizing a configuration that implements these path preferences is *at least as hard* as only synthesizing the path p_a in the first place.

Routing Algorithm	A	$\oplus$	$\preccurlyeq$	$\bullet$	C
Minimum Hop	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \{1\}$
OSPF	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \{1, 2, \dots, k; \infty\}$
Shortest Path	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \mathbb{R}^+ \cup \{\infty\}$
Most-Reliable Path	$[0, 1]$	$\times$	$\geq$	0	$E \to [0, 1]$

Table 6.2: *Examples of uniform routing algorithms that treat all destinations identically.*

6.1.3 Routing Algorithms

We introduce the abstraction of a *routing algorithm* defined as the combination of distributed, destination-based routing protocols that route traffic in a hop-by-hop manner. That is, each node forwards traffic according to its own local decision based solely on the packet's destination. We assume that nodes exchange routing information with their neighbors, and we require that they share at least their preferred route—which is the case for link-state and distance-/path-vector protocols.

We define a routing algorithm in terms of a Routing Algebra [106], that is, a mathematical tool that generalizes and models the computation of routing protocols. It has proven to be well-suited for protocol and configuration verification by capturing necessary conditions for optimal routing and eventual consistency [106, 108, 110, 111]. We transfer this concept to configuration synthesis by extending Routing Algebra with a notion of expressivity for an algorithm's configuration.

Formally, we define a *routing algorithm* as a five-tuple

$$\mathcal{A} = (A, \oplus, \preccurlyeq, \bullet, C)$$

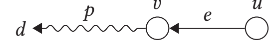
that contains both a Routing Algebra $(A, \oplus, \preccurlyeq, \bullet)$ as well as a description of all possible configurations C. A denotes the set of all attributes that includes the absorbing attribute $\bullet \in A$ to encode the absence of a route. Attributes are ordered by the partial order relation $\preccurlyeq$ and extended with the binary relation $\oplus$. Finally, a configuration $c \in C$ for a network $G = (V, E)$ with destinations (or prefixes) D is a function $c : (E \times D) \to A$ that assigns an attribute per destination to each edge.

Routing Algebra uses attributes A to describe both routes and weights on links, i.e., how routes are transformed.

We provide some common examples using this notation in Table 6.2. For instance, we model Shortest Path Routing as $\mathcal{A}_{SPR} = (\mathbb{R}^+, +, \leq, \infty, C_{SPR})$, based on the algebra of positive real numbers. A configuration $c \in C_{SPR}$ assigns a positive number, i.e., distance, to each edge. Nodes prefer the path with the smallest *cost*, i.e., the sum of all edge distances along the path.

Path Computation

We model propagated routes as tuples $(d, \sigma, p) \in D \times A \times \mathcal{P}$, containing a destination d, a cost σ, and a path p. Assuming node v prefers route (d, σ, p) to reach destination d, node v transforms and propagates the route $(d,\ c(e, d) \oplus \sigma,\ \langle e \rangle \circ p)$ towards u over the edge $e = (u, v)$. In other words, the attribute $\sigma \in A$ announced at $dst(p_{s,d})$ is transformed with the operator $\oplus$ (e.g., added up for SPR) on each edge of the path; i.e., $c(e_n, d) \oplus \ldots \oplus c(e_1, d) \oplus \sigma$ (right-to-left precedence). Node u then selects routes based on the partial order $\preccurlyeq$ over attributes. Further, u will route packets with destination d via e, i.e., the first edge in $\langle e \rangle \circ p$. The resulting forwarding graph for the destination d is a directed graph $F_d = (V, E_d)$ with out-degree at most 1, where $E_d \subseteq E$. A forwarding path in F_d from s to d is denoted as $p_{s,d}$, and we write $p_{s,d} \oplus_d \sigma$ to apply the label of each edge along $p_{s,d}$ in propagation order.



Routes propagate in the opposite direction of forwarding paths. Here, node v advertises its route for d to u over the (reversed) edge $e = (u, v)$.

Characteristics of Routing Algorithms

Recall that routing algorithms defines the attribute $\bullet$ to indicate the absence of a route. The following two properties assert that any route will always be preferred over the absence of a route. We assume that both properties hold for all routing algorithms.

Maximality: $a \prec \bullet$ for all $a \in A \setminus \{\bullet\}$.

Absorption: $a \oplus \bullet = \bullet \oplus a = \bullet$ for all $a \in A$.

Other properties that do not hold for all routing algorithms, however, are not as obvious. We call these *characteristics* of routing algorithms. To begin with, we adopt the notions of strict monotonicity and isotonicity from [106]:

Strict Monotonicity: $a \prec b \oplus a$ for all $a, b \in A \setminus \{\bullet\}$.

Strict Isotonicity: $a \prec b \Leftrightarrow c \oplus a \prec c \oplus b\ \forall a, b, c \in A \setminus \{\bullet\}$.

We find that, by themselves, these characteristics are not enough to analyze configuration synthesis. Consider Minimum-Hop Routing (MHR) $\mathcal{A}_{MHR}$ that routes traffic according to the path with the least number of hops (modeled as SPR with equal edge weights; cf. Table 6.2). Both $\mathcal{A}_{MHR}$ and $\mathcal{A}_{SPR}$ are identical regarding their routing algebra. Yet, $\textsc{Synth}(\mathcal{A}_{MHR}, \mathcal{S})$ degrades to computing the least-hop paths, whereas $\textsc{Synth}(\mathcal{A}_{SPR}, \mathcal{S})$ is solved using Linear Programming (LP).

Family of Routing Algorithms	Examples
Shortest Path Routing with finite bit representation	OSPF, IS-IS
Shortest Path Routing	SPR, MRPR
Routing with separate propagation	BGP with Route Reflection
Hierarchical routing	Hierarchical BGP, BGP Confederations
Non-uniform routing	Exclusively BGP, Static Routes

Table 6.3: *Classification of the most common routing protocol stacks.*

Consequently, we define three characteristics **C1**–**C3** to capture the expressivity of routing algorithms' configurations. Intuitively, a *uniform* routing algorithm routes traffic for all destinations identically; a *filtering* algorithm allows configuring, for each edge, whether a node accepts a route on this edge (a: allow) or not ($\bullet$: deny); and a *linear* routing algorithm is homomorphic to SPR. More formally:

C1 *Uniformity*: C requires any configuration $c \in C$ to have $c(e, d_1) = c(e, d_2)$ for all $e \in E$, $d_1, d_2 \in D$.

C2 *Filterability*: for some $a \in A : a \oplus b \prec \bullet$, $\forall b \in A \setminus \{\bullet\}$. Further, C allows both edge weights a and $\bullet$.

C3 *Linearity*: $\mathcal{A}$ is strictly monotone, strictly isotone, and there exists a mapping $g : \mathbb{R}_+ \to A$ such that:

1. $\forall a, b, c \in \mathbb{R}^+ : a = b + c \iff g(a) = g(b) \oplus g(c)$
2. $\forall a, b \in \mathbb{R}^+ : a < b \iff g(a) \prec g(b)$

For instance, consider Most-Reliable Path Routing (MRPR), where routers select the next hop according to the reliability of a path. MRPR can be denoted as $\mathcal{A}_{MRPR} = ([0, 1], \geq, \cdot, 0, C)$. MRPR is strictly monotone and isotone, filtering for any $a \neq 0$, and the function $g(x) = 1/e^x$ is a witness of its linearity.

A network running MRPR with link weights $c(e)$ will compute the same forwarding paths as the same network running SPR with weights $c^*(e) = -\log c(e)$.

Finally, we use the introduced notation and terminology to define five generic families of routing algorithms that cover the most common routing protocol stacks (cf. Table 6.3).

Shortest Path Routing (SPR) with finite bit representation. The most widely used intradomain routing protocols are based on SPR due to its simplicity and fast convergence upon network failures. They are uniform routing algorithms because link weights cannot discriminate different destinations. Further, they are strictly monotone and strictly isotone [106]. Practical limitations for their link weights, i.e., are represented with a finite number of bits, which makes them non-linear, while a special ∞-label makes them filtering.

Examples: OSPF, IS-IS, or any other real-world implementation of SPR.

Shortest Path Routing (SPR). Allowing arbitrary, real-valued link weights makes SPR not only a uniform but also a linear routing algorithm. This family includes all routing algorithms that are homomorphic to SPR (cf. Lemma 6.1), such as Most-Reliable Path Routing. Note that linear and uniform algorithms can only realize forwarding states that satisfy suboptimality (cf. Definition 6.5).

Examples: the idealized SPR and MRPR. Practical limitations make implementing such protocols infeasible.

Routing with separate propagation. iBGP can be thought of as distributing routing information on an overlay propagation graph. Techniques such as BGP Route Reflection are motivated by the poor scalability of maintaining iBGP sessions between all pairs of nodes to guarantee that each node eventually receives its preferred route. Nodes perform the actual routing with a different algorithm that is usually linear and uniform, e.g., an SPR algorithm. A configuration for such a routing algorithm involves finding both the overlay propagation graph and link weights for the SPR.

Example: iBGP Route Reflection where IGP paths are computed using shortest paths.

Hierarchical routing. Algorithms such as hierarchical BGP separate the network into multiple partitions. Each partition routes traffic internally, typically using a linear and uniform routing algorithm, but interconnects with neighboring domains using non-uniform routing. This hierarchy enables such an algorithm to compute forwarding states that do not satisfy suboptimality, as highlighted in the example in Figure 6.2.

Examples: multiple ASes or BGP Confederations, where each partition (either an AS or a confederation) is using SPR as an IGP.

Non-uniform routing. This family includes all non-uniform, filtering routing algorithms. They can independently implement any forwarding spanning tree for each destination prefix. Non-uniform routing is equivalent to hierarchical routing if every node constitutes its own partition.

Examples: routing solely using static routes or configuring eBGP on all links [173].

6.2 Complexity Analysis

Before exploring the complexity of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ for various families of routing algorithms $\mathcal{A}$, we define $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ and discuss how the three kinds of forwarding properties $\mathcal{S}$ relate to each other.

Definition 6.2 ($\textsc{Synth}(\mathcal{A}, \mathcal{S})$). *Given a graph $G = (V, E)$, a set of destinations D, and specification $\psi : (V \times D) \rightharpoonup \mathcal{S}$, does there exist a configuration $c \in \mathcal{C}$ for $\mathcal{A}$ such that $c \models \psi$?*

Observation 6.1. *Any path property $\psi(s, d) = \langle e_n, \dots, e_1 \rangle$ can always be expressed as a set of connected waypoint properties $\psi(src(e_i), d) = e_i$ for all $1 \leq i \leq n$. Thus, any set of path properties $\psi : (V \times D) \to \mathcal{P}$ can be expressed as a set of connected waypoint properties $\psi : (V \times D) \mapsto V \cup E$. Consequently, any intractability result for $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ with $\mathcal{S} = \mathcal{P}$ also applies for connected specifications, and a hardness result for a connected specification translates to unrestricted $\mathcal{S} = \mathcal{P} \cup V \cup E$. Likewise, any algorithm solving $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ for $\mathcal{S} = \mathcal{P} \cup V \cup E$ can also solve the problem if $\mathcal{S}$ is connected, and any algorithm for connected $\mathcal{S}$ can also solve the problem if $\mathcal{S} = \mathcal{P}$.*

The set of all path properties
$\subset$ the set of connected waypoints
$\subset$ the set of arbitrary waypoints.

6.2.1 Shortest Path Routing

This section explores the computational complexity of Shortest Path Routing, as well as its homomorphism, that is, a routing algorithm $\mathcal{A}$ for which each set of link weights $w \in \mathcal{C}_{SPR}$ for SPR can be transformed to an equivalent configuration $c \in \mathcal{C}$ for $\mathcal{A}$ and vice-versa. We call two configurations equivalent if they will always compute the same forwarding state under the same routing inputs. We first prove that any linear and uniform routing algorithm $\mathcal{A}$ is a homomorphism of SPR.

Lemma 6.1. *Any linear and uniform routing algorithm $\mathcal{A}$ is a homomorphism of SPR.*

Proof. Let $\mathcal{A}$ be a linear and uniform routing algorithm. We first show how we transform any configuration $c \in \mathcal{C}$ into equivalent link weights $w \in \mathcal{C}_{SPR}$ for SPR. Since $\mathcal{A}$ is strictly isotone, we can compute the All-Pairs-Shortest-Path (APSP) using the generalized Dijkstra algorithm [106]. While doing so, we construct a system of linear inequalities that we solve using LP to get the equivalent set of link weights w for SPR [174].

Next, we show how to transform any set of weights for SPR into equivalent attributes for $\mathcal{A}$. Let $w \in C_{SPR}$ be a set of link weights for SPR. We construct $c \in C$ for $\mathcal{A}$ by transforming each weight separately using $g(\cdot)$, i.e., $c(e) = g(w(e))$. Note that we neglect the destination d as both $\mathcal{A}$ and $\mathcal{A}_{SPR}$ are uniform.

The transformation function g refers to the witness of linearity from **C3***.*

We will now show that c is equivalent to w by relying on the Linearity of $\mathcal{A}$. Let $p = \langle e_n, \dots, e_1 \rangle$ be any path in G with $w(p) = w(e_n) + \dots + w(e_1)$. By repeatedly applying the first rule of **C3**, we get $g(w(p)) = g(w(e_n)) \oplus \dots \oplus g(w(e_1))$. Finally, let $s, t \in V$, let p^* be the shortest s-t-path, and let $p^- \neq p^*$ be any other s-t-path. $w(p^*) < w(p^-)$ implies that $g(w(p^*)) \prec g(w(p^-))$ by the second rule of g. Hence, any shortest path in G for w is also an optimal path under c as ties are broken identically. This proves Lemma 6.1. □

Lemma 6.1 allows us to analyze the class of *Synth*$(\mathcal{A}, \mathcal{S})$ problems for all linear and uniform algorithms $\mathcal{A}$ by solely considering SPR. Due to its importance in graph theory, SPR and its inverse problem have been studied extensively in the literature. Given a set of paths $\mathcal{P}$, the inverse SPR problem is finding link weights in a graph G such that each path $p \in \mathcal{P}$ from $src(p) = s$ to $dst(p) = t$ is the shortest s-t-path in G. Ben-Ameur et al. presented an algorithm to solve the inverse SPR problem in polynomial time using LP [171]. Consequently, *Synth*$(\mathcal{A}, \mathcal{S})$ for a linear and uniform $\mathcal{A}$ and $\mathcal{S} = \mathcal{P}$ is solvable in $\mathcal{O}((|V|^2 + |E|)^{2.5})$ time using LP [175].

However, most real-world implementations of SPR, like OSPF, differ from SPR as they represent link weights using a finite number of bits. Related literature shows that both finding an exact solution to the discrete inverse SPR problem is *NP*-complete [170], as well as finding a *good* approximation [176].

To the best of our knowledge, the computational complexity of the inverse SPR problem under waypoint constraints is still an open question. Call et al. have proven the inverse SPR problem *NP*-complete when some edges are required or forbidden on the shortest-path tree towards some destination [177]. Even though this problem does relate to *Synth*$(\mathcal{A}, \mathcal{S})$ for SPR with waypoints, a key difference is that the problem analyzed by Call et al. cannot require a specific source node to use an edge. Consequently, we analyze the complexity of *Synth*$(\mathcal{A}, \mathcal{S})$ for SPR with a connected specification containing waypoints.

Theorem 6.1. *For any linear and uniform routing protocol $\mathcal{A}$ and any connected specification $\mathcal{S} = \mathcal{P} \cup V \cup E$, deciding whether there exists a solution to* $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ *is* NP*-complete.*

We prove Theorem 6.1 by transforming One-In-Three 3SAT (X3SAT) to $\textsc{Synth}(\mathcal{A}, \mathcal{S})$, similar to the proof of Call et al., based on the fact that SPR cannot implement all forwarding graphs. We provide the proof in a technical report [10].

6.2.2 Routing with Separate Propagation

iBGP is an overlay signaling protocol to disseminate external routing information, while an IGP such as SPR performs the actual routing. In traditional iBGP, all nodes are connected in a full-mesh, leading to performance issues in larger networks [33]. Operators commonly use Route Reflection (RR) to reduce the number of iBGP sessions by selecting a small number of nodes, i.e., the route reflectors, that redistribute their preferred routes to all other nodes, i.e., the clients. Consequently, clients can only select their preferred route if at least one route reflector shares its preference. We consider the configuration of the overlay signaling protocol to be part of the synthesis problem.

The overlay signaling graph of BGP RR is a directed graph $G_{iBGP} = (V, E_{iBGP})$ with route propagation rules as described by Griffin et al. [135]. Operators can reduce the number of iBGP sessions by carefully designing G_{iBGP}. In the following, we thus consider the *iBGP Graph* problem: In order to find a valid propagation graph G_{iBGP} with a minimum number of iBGP sessions (i.e., edges E_{iBGP}), each node must receive its desired route along a valid propagation path in E_{iBGP} where all nodes of that path select the same route.

We require the network to be free of deflection. A packet can get deflected from its original path due to a router along the path selecting a different route. Such deflections can violate safety properties or even reachability by creating a forwarding loop and are difficult to detect.

Definition 6.3 (iBGP Graph). *An instance of the iBGP Graph problem is a tuple $I_{iBGP} = (V, D, \mathcal{R}, \psi', k)$, where V is a set of nodes, D is a set of destinations, $\mathcal{R} = (V \times D)$ is a set of routes, $\psi' : (V \times D) \mapsto \mathcal{R}$ captures route preferences, and $k \in \mathbb{N}$ is an upper bound on the number of iBGP sessions $|E_{iBGP}| \leq k$. A route $(v, d) \in \mathcal{R}$ originates at node v and announces destination d. A node v must select the route $\psi'(v, d)$ to reach the destination d. A solution to I_{iBGP} is a directed graph $G_{iBGP} = (V, E_{iBGP})$ with $|E_{iBGP}| \leq k$, such that, for all $v, d \in V \times D$, there exists a path $p \in G_{iBGP}$ from v to the origin of $\psi'(v, d)$ along which all nodes prefer $\psi'(v, d)$. More formally, we have $\forall u \in p : \psi'(u, d) = \psi'(v, d)$.*

$\textsc{Synth}(\mathcal{A}, \mathcal{S})$ for BGP RR while satisfying path properties requires solving I_{iBGP}. Indeed, we can transform a set of path properties ψ into ψ' by taking the path's destination, i.e., $\psi'(s, d) = (dst(\psi(s, d)), d)$. Thus, such a $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ is at least as hard as solving I_{iBGP}.

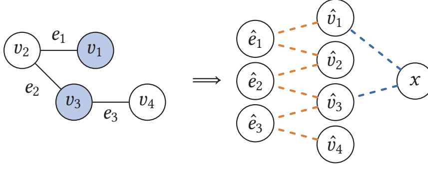


Figure 6.3: *An instance of Vertex Cover on the left is transformed into an equivalent iBGP problem on the right. Propagation sessions E_{iBGP} are drawn as dashed lines. Valid solutions to VC and iBGP are drawn in purple.*

Theorem 6.2. *Deciding if there exists a solution to an instance of the iBGP Graph problem is* NP*-complete.*

Proof. To begin with, we show that the iBGP Graph problem is in *NP* because a potential solution can be verified in polynomial time using a depth-first traversal.

We transform the *NP*-complete Vertex Cover (VC) problem to the iBGP Graph problem. VC is the problem of finding a set of nodes V^* of size $|V^*| \leq k'$ that includes at least one endpoint of each edge in graph $G' = (V', E')$. Let $I_{VC} = (G', k')$ be an instance of VC. Consider Figure 6.3 for an example of how to construct a basic instance of $I_{iBGP} = (V, D, \mathcal{R}, \psi', k)$ from I_{VC}. We first create the set of nodes V by taking each node in V' and each edge in E' and adding the auxiliary node x. We then construct the advertised routes $\mathcal{R}$ and the requirements ψ' to force iBGP sessions between nodes and their connected edges in G'. Finally, we require x to be connected to the smallest number of nodes V' to "cover" all edges E'. The following details this construction.

Let the nodes be $V = \{x\} \cup \{\hat{e} \mid e \in E'\} \cup \{\hat{v} \mid v \in V'\}$ and let $k = 2|E'| + k'$. Further, we create a destination d_e for each edge $e = (u, v) \in E'$, and let node $\hat{e}$ advertise d_e. In the following, pick any edge $e = (u, v) \in E'$. We show how we pick $\hat{\phi}$ such that any valid solution *(i)* must contain the orange session $(\hat{e}, \hat{u})$ and $(\hat{e}, \hat{v})$, and *(ii)* must either contain $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$, as drawn in blue in Figure 6.3. Then, we combine those two properties to conclude that I_{VC} is solvable if and only if I_{iBGP} has a solution.

To ensure that any valid solution contains the session $(\hat{e}, \hat{v})$ if $v \in e$, we create a new destination d_v which originates at every node $\hat{u} \neq \hat{v}$. Then, we require each node $\hat{u}$ to prefer the route it receives, while $\hat{v}$ prefers the route at $\hat{e}$, i.e., $\psi'(\hat{v}, d_v) = (\hat{e}, d_v)$, as visualized in Figure 6.4. Session $(\hat{e}, \hat{v})$ must exist, as $\hat{v}$ can only learn its desired route from $\hat{e}$. This requires any solution of I_{iBGP} to contain at least $2|E'|$ sessions.

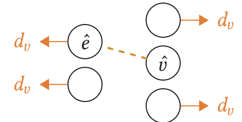
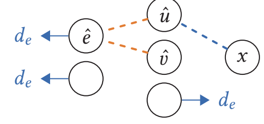


Figure 6.4: *Any solution must contain a session between $\hat{v}$ and $\hat{e}$ as $\hat{v}$ can only learn its route from $\hat{e}$.*

Next, we enforce any valid solution to contain either the session $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$. For each edge $e = (u, v) \in E'$, we create a destination d_e that is advertised at each node $w \in V \setminus \{\hat{u}, \hat{v}, x\}$. We require that each such node w selects its own route, i.e., $\psi'(w, d_e) = (w, d_e)$. The remaining nodes $\hat{u}$, $\hat{v}$, and x must all select the route advertised to $\hat{e}$, i.e., $\psi'(\hat{u}, d_e) = \psi'(\hat{v}, d_e) = \psi'(x, d_e) = (\hat{e}, d_e)$. Consequently, the propagation path p from d_e towards x can only contain nodes $p \subseteq \{\hat{e}, \hat{u}, \hat{v}, x\}$; any solution must contain $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$.



Node x can learn its route to d_e either from $\hat{u}$ or $\hat{v}$.

Finally, we show that I_{VC} has a solution if and only if I_{iBGP} is solvable. We first construct a solution V^* to I_{VC} from a solution $G_{iBGP} = (V, E_{iBGP})$ to I_{iBGP}. Due to the first property, E_{iBGP} contains at least $2|E'|$ sessions between any $\hat{e}_x$ and $\hat{v}_y$. Due to the second property, for each edge $e \in E'$, at least one of $\hat{e}$, $\hat{u}$, or $\hat{v}$ are neighbors of x. Now, let V^* be the neighbors of x in G_{iBGP}. If $\hat{e}$ is a neighbor of x for $e = (u, v)$, then simply add u to V^*. Since x must receive $(\hat{e}, d_e)$ for each edge $e \in E'$, V^* forms a vertex cover of G' and $|V^*| \leq k'$. Next, we show how to construct a solution to I_{iBGP} from a solution V^* to I_{VC}. Let E_{iBGP} contain session $(\hat{v}, x)$ for all $v \in V^*$, and session $(\hat{e}, \hat{u})$ for any edge $e \in E'$ with $u \in e$. $G_{iBGP} = (V, E_{iBGP})$ solves I_{iBGP} as $|E_{iBGP}| \leq k$ and V^* is a vertex cover of G', which causes x to receive the route $(\hat{e}, d_e)$ for each edge $e \in E'$. The construction can be done in polynomial time as I_{iBGP} contains $\mathcal{O}(|E'| + |N'|)$ nodes and destinations. This concludes the proof of Theorem 6.2. ☐

6.2.3 Hierarchical Routing

In hierarchical routing, operators partition their network, and each partition behaves as an individual AS. Each partition uses a linear and uniform routing algorithm such as SPR for routing traffic inside the domain, whereas a non-uniform algorithm like BGP exchanges routing information between domains. The introduction of non-uniform link weights increases the number of realizable forwarding states, i.e., the expressivity.

Formally, a routing algorithm $\mathcal{A}$ is hierarchical if *(i)* there exists a subset $A_i \subseteq A$ for each partition i such that $A_i, \oplus, \preccurlyeq, \bullet$ is strictly monotone and isotone, and *(ii)* edges of links within the partition i are uniform and chosen from A_i. In other words, for any edge $e \in E$ within partition i,

Links that connect two different partitions may have non-uniform weights chosen from the entire A.

$$\forall d_1, d_2 \in D : \underbrace{w(e, d_1) \in A_i}_{\text{monotone and isotone}} \quad \wedge \quad \underbrace{w(e, d_1) = w(e, d_2)}_{\text{uniform}}.$$

With a given partitioning of G, configuration synthesis for this partitioning is *at least as hard as* synthesizing configurations for the IGP used to route traffic within a partition. Otherwise, $\textit{Synth}(\mathcal{A}, \mathcal{S})$ also involves partitioning the network. Notice that assigning every node to its own partition essentially results in a non-uniform routing (cf. Section 6.2.4), as all links connect different domains. Yet, operators usually try to reduce the number of edges between partitions, i.e., minimizing the number of cut edges, as these tend to require significantly more maintenance. Thus, we require the solution of $\textit{Synth}(\mathcal{A}, \mathcal{S})$ to minimize the number of cut edges.

The following proves that $\textit{Synth}(\mathcal{A}, \mathcal{S})$ for any partitioned $\mathcal{A}$ is intractable for path properties $S = \mathcal{P}$. Since the link weights within a partition are uniform, strictly monotone, and strictly isotone, the forwarding paths inside this partition satisfy suboptimality [106]. We thus formulate the Partitioning into Suboptimal Paths (PSP) problem. Given a graph G and a set of paths P, find a partitioning of G such that the sub-paths of P contained within each partition satisfy suboptimality. We prove PSP to be *NP*-complete, thus proving that $\textit{Synth}(\mathcal{A}, \mathcal{S})$ is *NP*-hard for any partitioned $\mathcal{A}$.

Suboptimality requires any sub-path of an optimal path to be itself optimal.

Definition 6.4 (PSP). *An instance of PSP is $I_{PSP} = (G, P, j)$ with a directed multi-graph $G = (V, E)$ and a set of paths P. The partitioning $V_1, V_2, \ldots$ is a solution to I_{PSP} if and only if it cuts at most j edges and if the sets of sub-paths P_i of P contained within each partition V_i are suboptimal.*

Definition 6.5 (Suboptimality of Paths). *A set of paths P satisfies suboptimality if and only if for all pairs of paths $p_1, p_2 \in P$ that have two vertices u, v in common, the two paths share the same sub-path between u and v.*

Definition 6.6 (Contained Sub-paths). *The set of sub-paths P_i of P contained within V_i is the set of all sub-paths of P containing only edges with both endpoints in V_i.*

Theorem 6.3. *Deciding if there exists a solution to an arbitrary instance of PSP is* NP*-complete.*

Proof. To begin with, PSP is in *NP* because a potential solution to an instance of PSP can be verified in polynomial time by checking that all sub-paths contained within any partition satisfy suboptimality.

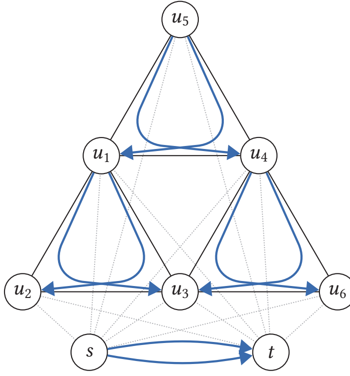


Figure 6.5: *For constructing an instance of PSP from an instance of NAE3SAT, we add paths for each clause (u_a, u_b, u_c) that will violate suboptimality if all nodes are in the same partition. This figure shows an example construction with three clauses (u_1, u_2, u_3), (u_5, u_1, u_4), and (u_4, u_3, u_6).*

We now transform the *NP*-complete problem Not-All-Equal 3SAT (NAE3SAT) to PSP. An instance of NAE3SAT is a tuple $I_{NAE3SAT} = (U, C)$, where U is a set of variables, and C is a set of clauses, each containing three variables. A solution is a truth assignment $U_t \subset U$ such that there does not exist a clause for which all variables have an equal assignment [178].

Let $I_{NAE3SAT} = (U, C)$ be an instance of NAE3SAT. We now construct $I_{PSP} = (G, P, j)$. The main idea is to create nodes for each variable and create paths for each clause that satisfy suboptimality only if all three nodes are not part of the same partition (cf. Figure 6.5). Let $G = (V, E)$ be a directed multi-graph with the set of nodes $V = U \cup \{s, t\}$ containing all variables and two special nodes s and t. We construct the edges E as follows:

- We add two edges $e^1_{s,t}$ and $e^2_{s,t}$ from s to t.
- For each variable $u \in U$, we add edge $e_{s,u}$ from s to u, and $e_{u,t}$ from u to t (dotted lines in Figure 6.5).
- For each clause $C_k = (u_a, u_b, u_c) \in C$, we add six edges that form a directed complete graph between u_a, u_b, and u_c called $e^\star_{a,b}$, $e^\star_{a,c}$, $e^\star_{b,a}$, $e^\star_{b,c}$, $e^\star_{c,a}$, and $e^\star_{c,b}$. The placeholder $\star$ in $e^\star_{x,y}$ is replaced by the number of already existing edges from u_x to u_y. The first edge from u_x to u_y will be called $e^0_{x,y}$, and the second $e^1_{x,y}$, etc.

The first edge $e^0_{x,y}$ from u_x to u_y exists if and only if there exists a clause that contains u_x and u_y.

Next, we construct the set of paths P as follows:

- We add two s-t-paths $\langle e^1_{s,t} \rangle$ and $\langle e^2_{s,t} \rangle$ to P.
- For each clause $C_k = (u_a, u_b, u_c) \in C$, we add the two paths $p^1_k = \langle e^0_{a,b}, e^0_{b,c} \rangle$, and $p^2_k = \langle e^0_{a,b}, e^0_{b,c} \rangle$ to P.

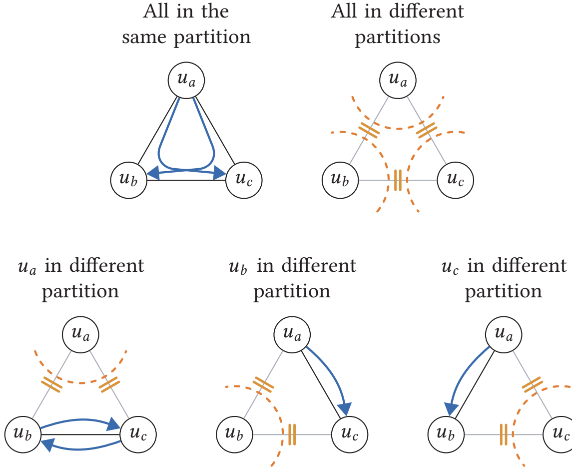


Figure 6.6: All five possibilities for partitioning the three nodes of a single clause $C_i = (u_a, u_b, u_c) \in C$ satisfy suboptimality, except if all nodes are in the same partition. This figure shows the resulting paths, as well as the cut edges.

Finally, we choose $j = 2 + |U| + 4|C|$. Let P_i be the set of sub-paths of P contained within partition V_i. There are five possibilities for how the partitioning can split the nodes of any clause $C_k = (u_a, u_b, u_c) \in C$ into partitions (cf. Figure 6.6):

1. All nodes are part of the same partition V_i, violating suboptimality as both paths remain uncut, i.e., $p_k^1, p_k^2 \in P_i$.
2. All three nodes are in different partitions. The remaining sub-path of p_k^1 and p_k^2 contained within any of the three partitions has zero sizes.
3. Two of the three nodes are in the same partition, resulting in the sub-paths of p_k^1 and p_k^2 having a length of at most 1. Further, for any pair of nodes $u_x, u_y \in V_i$, there can only be one path from u_x to u_y in P_i, which is $\langle e_{x,y}^0 \rangle \in P_i$.

Consequently, any solution to I_{PSP} cannot assign the same partition to all nodes of any clause. Further, if at least one node is part of a different partition for all clauses, then all sub-paths within each partition will satisfy suboptimality.

We will now show that any solution to I_{PSP} partitions the network into exactly two partitions, such that the number of cut edges is less than $j = 2 + |U| + 4|C|$. For this, consider the clause $C_k = (u_a, u_b, u_c) \in C$. Exactly four edges will be cut unless all three nodes are in different partitions, in which case all six edges are cut (cf. Figure 6.6). Further, we know that s and t are part of two different partitions, as $\langle e_{s,t}^1 \rangle$ and $\langle e_{s,t}^2 \rangle$ are not suboptimal. Consequently, the optimal partitioning

that solves I_{PSP} will use two partitions and will cut precisely $j = 2 + |U| + 4|C|$ edges; two edges connecting s and t, i.e., $e^1_{s,t}$ and $e^2_{s,t}$, then either edge $e_{s,u}$ or $e_{t,u}$ for each node $u \in U$, and finally, four edges for each clause. Hence, there are two clusters, and no clause has all nodes in the same cluster, which is analogous to the NAE3SAT problem.

We now show that $I_{NAE3SAT}$ has a solution if and only if I_{PSP} is solvable. Let $V_1, V_2, \ldots$ be a solution to I_{PSP}. The truth assignment $U_t = U \cap V_1$ is also a solution to $I_{NAE3SAT}$ because *(i)* $V_2 = V \setminus V_1$ since any solution to I_{PSP} has precisely two partitions, and *(ii)* for each clause, the three nodes cannot all be in the same partition. Similarly, let the truth assignment U_t be a solution to $I_{NAE3SAT}$. Then, the partitioning $V_1 = U_t \cup \{s\}$ and $V_2 = V \setminus V_1$ is a solution to I_{PSP}, as the nodes of any clause must not have equal value, i.e., are not all in the same partition. The construction is polynomial in time, as there are $\mathcal{O}(|U|+|C|)$ edges in G. This proves Theorem 6.3. ☐

6.2.4 Non-Uniform Routing

Some routing protocols, most notably BGP, can be configured without limitations. Operators can configure route maps to modify route attributes arbitrarily for each destination prefix, or the route can even be blocked entirely. A practical synthesis tool for configuring data centers exclusively using BGP already exists [87]. Yet, the computational complexity of configuring a network only with BGP is yet to be analyzed. We model such protocols as filtering and non-uniform algorithms. The filtering property allows nodes to prefer specific edges over others, while the non-uniformity enables the network to apply this preference separately per destination. These two properties allow us to construct a configuration to achieve any forwarding forest F_d for each destination $d \in D$ by picking link weights appropriately. We then use this fact to analyze the SYNTH$(\mathcal{A}, \mathcal{S})$ problem for different kinds of specifications.

Lemma 6.2. *Let $G = (V, E)$ be a graph with destinations D, and let $\mathcal{A}$ be a non-uniform and filtering routing algorithm. There exists a configuration $c \in C$ for $\mathcal{A}$ that results in any spanning forest $F_d = (V, E_d)$ for destination $d \in D$.*

Proof. We construct a $c \in C$ that computes the spanning forest $F_d = (V, E_d)$. Let a be a weight such that $a \oplus b \prec \bullet$ for all $b \in A \setminus \{\bullet\}$ as required by the Filtering property **C2**. We construct c as follows:

$$c(e,d) = \begin{cases} a & \text{if } e \in E_d \\ \bullet & \text{if } e \notin E_d \end{cases}$$

We now show that c computes F_d. Let $p = \langle e_n, \dots, e_1 \rangle$ be a path, and let $e_i \notin E_d$ be any edge in p which is not in F_d. Further, let p^* be the path on F_d from $s = src(p)$ to its root. We will now show that node s will prefer $p^* \oplus_d \sigma_1$ over $p \oplus_d \sigma_2$ for any $\sigma_1, \sigma_2 \in A \setminus \{\bullet\}$. $p^* \oplus_d \sigma_1 = a \oplus \cdots \oplus a \oplus \sigma_1 \prec \bullet$, as any edge in p^* is on T_d. However, by absorption, $p \oplus_d \sigma_b = \cdots \oplus \bullet \oplus \cdots \oplus \sigma_b = \bullet$. Consequently, node s will always prefer $p^* \oplus_d \sigma_1$ over $p \oplus_d \sigma_2$. This proves Lemma 6.2. □

Most non-uniform algorithms $\mathcal{A}$ can implement any spanning tree without filtering routes. BGP, for instance, can implement a given spanning tree for destination d by assigning routes learned over edge $e \in E_d$ a high local preference. This ensures the network maintains reachability under link failures.

We now consider specifications containing arbitrary (non-connected) waypoints. We first define the Spanning Tree with Waypoints (STW) problem: find a forwarding spanning tree F_d for destination d that satisfies waypoint properties. By proving STW to be *NP*-complete, we also show that *Synth*($\mathcal{A}, \mathcal{S}$) is *NP*-hard as we can transform STW to *Synth*($\mathcal{A}, \mathcal{S}$) by considering a single destination d. Perhaps more surprisingly, Theorem 6.4 proves *Synth*($\mathcal{A}, \mathcal{S}$) to be *NP*-hard for *any* filtering algorithm that can implement any spanning tree for a single destination.

Definition 6.7 (Spanning Tree with Waypoints). *An instance of the STW problem is a tuple $I_{STW} = (G, r, \psi')$, where $G = (V, E)$ is a graph, $r \in V$ is a root node, and $\psi' \in (V \times V)$ is a set of pairs of nodes. A solution to the STW problem is a spanning tree $T = (V, E_T)$ rooted at r such that for all pairs $(s, \omega) \in \psi'$, the path from s to r traverses node ω.*

Theorem 6.4. *Deciding whether an arbitrary instance of STW is solvable is* NP*-complete.*

Proof. To start, we show that STW is in *NP*. This is because we can verify a potential solution in polynomial time by traversing the forwarding tree from each node to the root r.

We now transform Disjoint Connecting Paths (DCP) to STW. DCP is the *NP*-complete problem of finding a set of k vertex disjoint paths in a graph $G = (V, E)$ that connect each of the k pairs in $x \in V \times V$ [178].

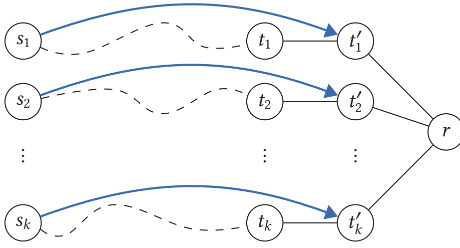


Figure 6.7: For constructing an instance of STW from DCP, we add the root node r, as well as nodes t'_i, such that a spanning tree from a solution to STW can be formed by connecting each t_i to t'_i, and t'_i to r. The arbitrary waypoint properties are shown as red arrows.

Let $I_{DCP} = (G, x)$ be an instance of the DCP problem. We construct a problem instance $I_{STW} = (G', r, \psi')$ that has a solution if and only if I_{DCP} has a solution, cf. Figure 6.7. Let $G' = (V', E')$ be a graph constructed from G, adding the root node r and, for each pair $(s_i, t_i) \in x$, the node t'_i that is only connected to t_i and r. Finally, we define the waypoint properties that s_i must reach r via t'_i, i.e., $\psi' = \{(s_i, t'_i) \mid (s_i, t_i) \in x\}$.

We now show how to construct a solution to I_{DCP} from a solution to I_{STW} in polynomial time. Let $T = (V, E_T)$ be a spanning tree that solves I_{STW}, which means that all waypoint properties are satisfied. Since T is a tree, there can only be one path from s_i to r, which we call p'_i. Each path p'_i must end in $\langle (t_i, t'_i), (t'_i, r) \rangle$ because $t'_i \in p'_i$ and t'_i has precisely two neighbors, t_i and r. r is the root node and can, therefore, not be a child of t_i. Further, any two paths in a tree from two nodes u and v to the root can merge but not separate since any node has at most one parent. Hence, all paths from s_i to r are vertex disjoint. As a consequence, the set of sub-paths from p'_i up to node t_i form a solution to I_{DCP}.

Finally, we use a solution to I_{DCP} to construct a spanning tree $T = (V, E_T)$ that solves I_{STW} in polynomial time. Let $p_1, p_2, \ldots$ be a set of vertex-disjoint paths, where $\forall (s_i, t_i) \in x$, p_i is a path from s_i to t_i. We first add all edges in all paths to E_T, i.e., $\forall i : p_i \subseteq E_T$. We then set the parent of each t_i to t'_i, and the parent of t'_i to r, i.e., $(t_i, t'_i) \in E_T$ and $(t'_i, r) \in E_T$. Finally, we connect all remaining nodes arbitrarily to r using a breadth-first traversal that preserves the tree. T is a spanning tree because *(i)* each node is connected to the root node, and *(ii)* every node except r will have precisely one parent as every node is traversed at most once by all paths p_i. The spanning tree satisfies all waypoints in ψ' because each t_i has t'_i as its parent. This concludes the proof of Theorem 6.4. □

Data: $G = (V, E)$, destination d, routes ρ, and specification ψ
Result: Spanning forest $F_d \models \psi$ that satisfies the specification.

```
F_d ← ∅
V_0 ← {v | v ∈ ρ : ⟨⟩ ⊨ ψ(v, d)}                       Initialize all roots.
q ← {(src(e), ⟨e⟩) | e ∈ E : dst(e) ∈ V_0}             Initialize the queue
while q ≠ ∅ do
    (v, p = ⟨e_n, …, e_1⟩) ← q.pop()
    if v ∉ F_d ∧ p ⊨ ψ(v, d) then      p must satisfy the specification
        F_d(v) ← e_n                                   Set v to use path p
        foreach e ∈ E : dst(e) = v do
            q.push((src(e), ⟨e⟩ ∘ p))  Enqueue incoming edges of v
if ∃v ∉ V_0 : F_d(v) = ∅ then
    raise unsat
```

***Algorithm 6.1**: Generate a spanning forest F_d for destination d that satisfies the specification ψ. This algorithm is based on a breadth-first traversal. Note that we traverse the tree in the opposite direction of the forwarding paths.*

Connected Waypoints

We now present an algorithm to synthesize configurations for any filtering and non-uniform routing algorithm $\mathcal{A}$ that satisfies connected specifications in polynomial time. The algorithm performs a breadth-first traversal for each destination $d \in D$ to build the spanning forest F_d. This traversal starts at the set of nodes $\rho(d) \subseteq V$ that originate the destination d. The algorithm explores node v only if the current path p satisfies the property $p \models \psi(v, d)$. We generate a configuration c for destination d using Lemma 6.2 and the constructed spanning forest F_d. The algorithm shown in Algorithm 6.1 has running time $\mathcal{O}(|D| \cdot |E|)$, as it performs one breadth-first traversal (with complexity $\mathcal{O}(|E|)$) for each $d \in D$.

This polynomial-time algorithm proves that $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ for any filtering and non-uniform $\mathcal{A}$ and connected specifications $\mathcal{S}$ is solvable in polynomial time.

Theorem 6.5. *For any graph $G = (V, E)$, connected specification ψ, and destination $d \in D$ announced at $\rho \subseteq V$, Algorithm 6.1 yields a valid spanning forest for d that satisfies ψ if and only if such a spanning forest exists.*

Proof. We first show that Algorithm 6.1 constructs a spanning forest $F_d \models \psi$ that satisfies ψ. This is the case because the algorithm assigns a parent only if the resulting path satisfies its property. Otherwise, if no valid spanning forest exists, then the algorithm raises *unsat*.

We now show that Algorithm 6.1 will find a valid solution if there exists a solution $F_d^* \models \psi$. For the sake of contradiction, assume it fails to find a spanning forest and raises *unsat*. Thus, there exist non-root nodes $v \notin V_0$ without parents in F_d. Let V_x be the set of such nodes, and pick any $v \in V_x$. By Definition 6.1, there exists a path p from v to a root in V_0 along with the property of each node complies with $\psi(v, d)$. Let e be the last edge on this path p for which $dst(e) \in V_x$, but $src(e) \notin V_x$. By Definition 6.1, the sub-path of p starting at $src(e)$ complies with $\psi(src(e), d)$. Thus, e is added to F_d, contradicting the assumption that $src(e) \in V_x$ and proving Theorem 6.5. □

Recall that Definition 6.1 defines connected waypoints if there exists a path from a node to its waypoint ω that traverses only other nodes with the same waypoint constraint ω or path constraints through ω.

6.3 Related Work

Before concluding this chapter, we summarize the literature related to formal methods applied to network management.

Network configuration synthesis. Recently, the literature has proposed several systems to synthesize configurations. Many aim to be as general as possible, allowing operators to deploy them in a wide range of networks [29, 84, 85]. Unfortunately, these systems do not scale to large networks. For instance, the state-of-the-art SMT-based system *NetComplete* takes around 6 hours to synthesize configurations for 64 nodes [29].

Other synthesizers specifically target data centers, allowing these systems to utilize domain-specific knowledge to improve their scalability with a modular synthesis approach [28, 87]. While being very effective in synthesizing configurations for data-center networks, their approach has yet to be generalized for other Internet networks, e.g., transit networks.

Other systems follow a best-effort approach and use static routes to resolve any inaccuracies [30], which enables much better scaling than for SMT-based approaches; however, without providing guarantees to find a solution and at the cost of network reliability, e.g., upon failures.

Finally, some synthesis tools rely on Large Language Models (LLMs) to generate configurations [88–92]. To avoid syntactic and semantic errors, they rely on a verifier and/or a human operator and rely on their feedback for the next iteration. While significantly improving synthesis times, this approach cannot guarantee that they will eventually find a configuration or that their process will terminate if no correct configuration exists.

In contrast, this paper does not propose a system to facilitate configuration synthesis in a specific scenario but explains the limitations of previous work by studying the computational complexity of configuration synthesis.

Network configuration verification. Some verification systems use SMT-based solvers to check both forwarding and control-plane properties under different failure scenarios [23, 48, 65, 66]. Modern approaches show great scalability by relying on a modular and concurrent design [73–75]—this promising insight may also apply to configuration synthesis. However, even though verification and synthesis seem very similar, there are critical differences between them. For example, verification aims at analyzing unstructured and human-written configurations that grew over the years, whilst synthesis can limit itself to a well-defined and structured configurations. Thus, our tractability results do not immediately apply to Configuration Verification. For instance, while configuration synthesis generally gets easier with increasing expressivity, verification becomes harder.

BGP is Turing-complete [179]. The verification problem thus becomes undecidable. Despite that, synthesizing configurations for a network routing traffic only using BGP can be done in polynomial time.

Complexity of Configuration Synthesis. The computational complexity of individual aspects of configuration synthesis has already been analyzed in the literature. Several studies have focused on inverting Shortest Path Routing [170, 171, 174, 176]. In this paper, we analyze the problem of computing link weights for SPR to realize a set of waypoint properties. Other aspects, like adding or removing individual iBGP sessions [180] or partitioning the network to comply with BGP propagation rules [30], were already shown to be intractable.

6.4 Conclusion

In this chapter, we analyzed the computational complexity of configuration synthesis by exploring the spaces of both routing protocols and forwarding properties. To that end, we identify a set of fundamental forwarding properties that *(i)* capture the synthesis problem to transition from being computationally tractable to *NP*-hard and *(ii)* allow our results to generalize to many higher-level requirements. We find that synthesizing configurations satisfying arbitrary waypoint constraints is *NP*-hard for any hop-by-hop routing protocol.

We further model routing protocols and their combinations as abstract *routing algorithms* and characterize these algorithms according to the properties of their routing computation. We find the complexity of configuration synthesis to depend on the expressivity of the routing algorithm, that is, how freely its parameters can be tuned. For instance, we can synthesize link weights for SPR that satisfy path properties in polynomial time only if these weights can take any positive real number. Next, we find that synthesizing configurations for uniform routing protocols (i.e., protocols using a shared routing configuration for all destination prefixes) is generally *harder* than for non-uniform protocols, as the latter can realize forwarding graphs for each destination independently. However, increasing the expressivity of a configuration does not necessarily imply that its synthesis becomes *easier*. To see that, consider SPR and hierarchical BGP; the latter is more expressive as the former can only realize forwarding states that satisfy the suboptimality principle. Checking for the existence of an SPR configuration that meets a given specification is computationally tractable, while the same problem for hierarchical BGP is *NP*-hard.

We investigated the complexity of configuration synthesis for the most common routing algorithms, but we could not analyze *all* of them. For example, the complexity of synthesizing configurations for Widest-Path Routing (e.g., EIGRP), as well as other algorithms that lack strict monotonicity and strict isotonicity, is still an open question. Further, we did not investigate network specifications beyond properties on forwarding paths, as we found these to be already challenging enough to satisfy. We expect the problem to become more difficult when considering dynamic networks that experience failures and changes in routing inputs.

7 Conclusion and Outlook

In this dissertation, we present four endeavors to extend the capabilities of formal methods in network operations, bringing such systems closer to Internet-wide adoption. In particular, we enable novel use cases, analyze new networking properties, and support additional routing protocols in network verification, configuration synthesis, and reconfiguration tools.

In Chapter 3, we present the *BGP State Iterator*, an algorithm to directly explore the space of BGP routing states. Doing so enables users to *guide* their exploration, either directly by the specification, by how relevant they are to operators, or by the probability of observing such states in the future. Specification-guided exploration reduces the search space significantly, finding all counterexamples within seconds, whereas related works timeout after one day. Further, the number of reachable BGP routing states tends to be manageable, even when receiving dozens of routes and thousands of possible propagation paths.

The *BGP State Iterator* enables new verification use cases, enabling operators to directly manipulate the routing state exploration, e.g., to find all counterexamples in decreasing order of their relevance or likelihood. In doing so, the *BGP State Iterator* sets the stage for probabilistic control-plane verification over failures and BGP routing events.

In Chapter 4, we present *Velo*, the first verifier to reason about performance properties under both link failures *and* BGP routing changes. Specifically, *Velo* focuses on worst-case link load properties, supporting operators in complex tasks, from improving configurations to business decisions. By proving link loads reach their worst case in a tiny subset of possible BGP routing events, we find worst-case link loads within minutes for large networks, even considering up to two link failures.

Velo extends network verification techniques to reason about performance-related properties that relate to all traffic in the network, compared to previous tools that focus on the forwarding or propagation path of individual destinations.

In Chapter 5, we present *Chameleon*, a system to safely and practically reconfigure any BGP network. *Chameleon* finds one convergence sequence that satisfies a given specification and enforces this sequence using temporary BGP configuration commands. This design enables safe BGP reconfigurations without noticeable overhead, especially compared to related works duplicating the entire control plane. To that end, we formulate a scheduling problem as an ILP that can be solved within minutes while finding reconfiguration plans that *Chameleon* applies within minutes.

Chameleon extends reconfiguration systems to support BGP and introduces a general approach to perform reconfigurations of distance- and path-vector protocols in-place.

In Chapter 6, we systematically analyze the computational complexity of network-wide configuration synthesis. We do so by investigating how interactions between routing protocols and forwarding properties influence the tractability of network configuration synthesis. We find that the expressiveness of a protocol's configuration influences the complexity: generally speaking, expressive protocols tend to be easier to synthesize configurations for. Regarding specifications, we prove that satisfying arbitrarily complex waypoint properties is generally *NP*-hard for any hop-by-hop protocols. Nevertheless, we find that most higher-level intents result in structural waypoint properties that are easier to find configurations for.

Our complexity analysis shows how carefully choosing the routing protocols to implement different network properties can yield scalable configuration synthesis tools.

7.1 Open Problems

We identify several opportunities for future work in formal methods applied to network operations. The following presents a subset regarding verification, synthesis, and reconfiguration. We focus on the fundamental ideas that further push formal methods in network operations while omitting straightforward ones, like improving the systems presented in this dissertation regarding their scalability and support for more aspects of the routing protocols.

7.1.1 Towards a Probabilistic Model for Routing Events

One goal of network verification is to prevent networks from misbehaving in future environments, e.g., under link failures or routing events. However, without knowing how *likely* such environments are to appear in the future, verification systems tend to find problems that are very unlikely to appear in the coming weeks or months [117]. Most network verifiers already capture the probabilities for failure scenarios, either implicitly by allowing up to k link failures or explicitly by modeling each link's failure rate, including dependencies from Shared Risk Link Groups (SRLGs) [23, 133]. However, such a probabilistic model for BGP routing events does not yet exist.

As a first step, *Velo* restricts the number of routing changes and, thus, implicitly models the likelihood of routing events. However, it assumes that *(i)* received routes are independent of each other and *(ii)* any change to routing information is equally likely. Both of these assumptions do not hold in BGP. For instance, it is *(i)* likely that routes with similar AS paths are affected by the same event and *(ii)* unlikely that a customer advertises its prefix with a longer AS path than a provider.

This is similar to considering up to k simultaneous link failures, but applied to routing events.

We see two promising approaches for designing a model that capture the probability of BGP routing events. We stress that many use cases do not require a perfectly accurate model. A simple one suffices to find likely counterexamples.

Data-driven approach. We conjecture that past routing events can predict likely future events (at least to a certain degree). Powerful machine-learning techniques might be able to find and exploit patterns hidden in historical BGP advertisements. Fortunately, training data is readily available as many operators collect historical BGP advertisements, and some make them publicly accessible [181, 182].

Model-driven approach. Failures, planned reconfigurations, and peering changes in external networks trigger routing events. These events likely impact most (or all) routes that traverse the affected AS. We envision modeling routing events as changes in an inter-AS topology. The probability of receiving a set of BGP updates can be approximated by how many such topology changes are necessary. We can pair this approach with the data-driven approach by learning which changes are likely and which prefixes or ASes are affected regularly.

7.1.2 *Towards General Performance Verification*

Velo presents first steps towards verifying network performance properties under routing inputs by focusing on worst-case link loads. However, it cannot directly be generalized to verify other performance aspects such as delay or bandwidth. We expect analyzing such performance properties to be more challenging than finding worst-case link loads. Indeed, delay and bandwidth become interesting properties to analyze in an overutilized network in which congestion has non-linear effects on the network performance. In contrast, finding worst-case link loads pertains to checking if congestion can occur in the first place.

Beyond worst-case performance. *Velo* focuses on the worst-case link loads while allowing the operator to restrict the search space to consider probable environments. Despite that, *Velo* likely overestimates link loads that the network will experience in the future. We see opportunities in answering probabilistic queries, such as finding link loads that are exceeded only with a 5% probability over routing events. We conjecture that our techniques from both *Velo* and the *BGP State Iterator* can be used to tackle probabilistic performance analysis as well.

7.1.3 *Towards Scalable Network Configuration Synthesis*

Faced with the intractability of most synthesis problems we derived in Chapter 6, it appears that configuration synthesis is unlikely to become widely deployed. On the contrary, we believe our results can guide future synthesis systems to achieve scalability. In fact, many existing synthesis tools [29, 30, 84] tend to use (or abuse) routing protocols for problems they were never designed to solve. For instance, these systems tweak link weights for OSPF to implement arbitrary waypoints instead of using Segment Routing—a technique that forwards packets along waypoints, regardless of failures and route changes.

We envision promising future synthesis systems that follow three ideas. First, the specification should describe all aspects of the network to give the operators enough fine-grained control. Second, they should follow predictable rules to generate the low-level router configurations that match the users' expectations. Finally, synthesis should not focus on problems already solved by the network protocols but on optimizing network performance. We provide details for each idea in the following:

A programming language for describing networks. As in other verification domains [183], designing a precise specification is notoriously tricky, primarily because operators are used to different kinds of tools [117]. Network verification remains useful without a precise specification, verifying only those aspects that the operators can formally express. However, an imprecise specification makes a synthesizer generate configurations that contradict operator's intent while technically satisfying the specification. Thus, an accurate and expressive specification language targeting users less familiar with formal tools is vital for configuration synthesis.

We envision a declarative specification language to describe all aspects of a network configuration at a higher level, akin to existing domain-specific languages, like NetKAT for data-plane verification [52]. Such a specification must include the network topology, routing policies, security properties, and performance objectives, among others. The key challenge is navigating the tradeoff between the level of abstraction and its expressivity: how to simplify operations while allowing fine-grained control over network behavior.

Compiling network configurations. Using an expressive specification language, we conjecture that low-level router configurations can be *compiled* from a high-level specification without resorting to common synthesis techniques like SMT. We argue that more detailed specification language is desired for both network operators and synthesis tools. Indeed, reading such a concrete specification language *(i)* allows users fine-grained control over the generated configurations and *(ii)* avoids the intractable problems associated with synthesizing configurations from high-level properties.

We envision the compiler to check the consistency of the specification and for common pitfalls such as route leaks and a disconnected management network, both of which have caused recent large-scale outages [21, 184].

There exists numerous tools for regular programming languages to improve developer experience. We believe that these ideas translate to the networking context as well.

Propane [28, 87] presented first steps towards such a compiler, generating BGP configurations from specifications defined in their domain-specific language. *Propane* avoids the intractability of configuration synthesis by designing a compiler that ensures preferences for propagation paths by relying on graph algorithms, allowing it to generate network-wide configuration within minutes.

Finding an optimal network architecture. Today's Internet routers support a wide range of flexible routing protocols that provide various overlapping functions. These protocols come with different tradeoffs, for instance, requiring more memory usage or causing control-plane churn. Designing a network architecture, i.e., which protocols are used in which ways to perform which functions, is a challenging task that depends on many aspects of the network and its demands.

In essence, designing a network architecture entails solving an optimization problem: What combination of protocols and their features is *(i)* sufficient to implement the specification and *(ii)* supported by all routers deployed in the network that *(iii)* minimizes overhead, such as memory usage, control-plane churn, or performance degradation. We envision a technique to measure the overhead of a network architecture to provide insightful tools for network operators to design or redesign their network architecture. Such a framework would enable network operators to automatically find the optimal network architecture and to provide boundaries describing when to replace protocols.

7.1.4 Towards Safe and Fast BGP Convergence

Chameleon focuses on performing BGP reconfigurations safely. In doing so, we developed a framework to controllably update BGP, which is useful not only for reconfiguring networks safely but also for doing so *quickly*. Indeed, during convergence, BGP may explore different paths until the optimal one is found, unnecessarily disseminating suboptimal routes in the process. Our model guarantees that each router updates its selected route exactly once, from the initial to the final route, thus avoiding this BGP path exploration process.

Chameleon's reconfiguration time is limited by how fast it can reconfigure the routers BGP configuration. Yet, if given more direct control over the routing process, such as with *xBGP* [185, 186], we believe that future work should revisit *Chameleon* regarding reconfiguration time. Can *Chameleon*'s reconfiguration time (or its scheduling time) be minimized by relaxing its security guarantees, i.e., only maintain reachability?

Reacting to unexpected BGP events. Reconfiguration systems like *Snowcap* and *Chameleon* apply planned reconfigurations. Yet, convergence processes are also triggered by unexpected BGP events, likely causing transient blackholes and forwarding loops [6, 9]. Avoiding such anomalies is challenging due to BGP's the distributed and asynchronous nature. There has been significant progress towards speeding up BGP convergence [139, 169], but they either completely redesign BGP or fail to provide guarantees. We suspect that insights from *Chameleon* also apply to unexpected BGP events, for instance, by running a simplified instance of *Chameleon* on each router and altering route propagation to preserve reachability.

Own publications

[1] **Tibor Schneider**, Jean Mégret, and Laurent Vanbever. "Guided Exploration of Control-Plane Routing States". In: *2025 IEEE 33th International Conference on Network Protocols (ICNP)*. 2025.

[2] **Tibor Schneider**, Stefano Vissicchio, and Laurent Vanbever. "Verifying maximum link loads in a changing world". In: *22th USENIX Symposium on Networked Systems Design and Implementation*. 2025.

[3] **Tibor Schneider**, Roland Schmid, Stefano Vissicchio, and Laurent Vanbever. "Taming the transient while reconfiguring BGP". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023.

[4] **Tibor Schneider**, Roland Schmid, and Laurent Vanbever. "On the Complexity of Network-Wide Configuration Synthesis". In: *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. 2022.

[5] **Tibor Schneider**, Rüdiger Birkner, and Laurent Vanbever. "Snowcap: Synthesizing Network-Wide Configuration Updates". In: *Proceedings of the ACM SIGCOMM 2021 Conference*. 2021.

[6] Roland Schmid, **Tibor Schneider**, Georgia Fragkouli, and Laurent Vanbever. "Predicting Specification Violations During BGP Convergence". In: *Proceedings of the CoNEXT Student Workshop 2023*. 2023.

[7] Yu Chen, **Tibor Schneider**, and Laurent Vanbever. "Causality Analysis in Control Plane Verification". In: *Proceedings of the CoNEXT Student Workshop 2023*. 2023.

[8] Roland Schmid, **Tibor Schneider**, Georgia Fragkouli, and Laurent Vanbever. *The Effects of iBGP Convergence*. 2025. URL: *https://arxiv.org/abs/2503.15249*.

[9] Roland Schmid, **Tibor Schneider**, Georgia Fragkouli, and Laurent Vanbever. "Transient Forwarding Anomalies and How to Find Them". In: *Proceedings of the 21th ACM International Conference on Emerging Networking Experiments and Technologies*. 2025.

[10] **Tibor Schneider**, Roland Schmid, and Laurent Vanbever. *On the Complexity of inverting Shortest-Path Routing with Connected Waypoint Constraints*. 2022. URL: *https://doi.org/10.5281/zenodo.7040824*.

Bibliography

[11] Rachel King. *Here's Why Amazon's Cloud Suffered a Meltdown This Week*. Accessed: 2025-05-19. 2017. URL: *https://fortune.com/2017/03/02/amazon-cloud-outage/*.

[12] Doug Madory. *Facebook's historic outage, explained*. Accessed: 2021-03-09. 2021. URL: *https://kentik.com/blog/facebooks-historic-outage-explained/*.

[13] Jay Peters. *Facebook, Instagram, and Threads have recovered after a massive outage*. Accessed: 2025-05-19. 2024. URL: *https://www.theverge.com/2024/12/11/24318908/facebook-instagram-threads-down-outage-meta*.

[14] Queenie Wong. *Facebook's massive outage costs the company an estimated $60 million in revenue*. Accessed: 2025-05-19. 2021. URL: *https://www.cnet.com/tech/mobile/facebooks-massive-outage-costs-the-company-an-estimated-60-million-in-revenue/*.

[15] Mallory Locklear. *Google accidentally broke the internet throughout Japan*. Engadget. Accessed: 2025-05-19. 2017. URL: *https://engadget.com/2017-08-28-google-accidentally-broke-internet-japan.html*.

[16] Jon Mettler. *Ursache der landesweiten Swisscom-Störung ist bekannt*. Accessed: 2025-05-20. 2021. URL: *https://www.tagesanzeiger.ch/ursache-der-landesweiten-swisscom-stoerung-ist-bekannt-248254149993*.

[17] Tages-Anzeiger. *Swisscom erklärt die Ausfälle der Notfallnummern*. Accessed: 2025-05-19. 2021. URL: *https://www.tagesanzeiger.ch/notfallnummern-in-weiten-teilen-der-schweiz-ausgefallen-981652744443*.

[18] Kotikalapudi Sriram, Doug Montgomery, Danny R. McPherson, Eric Osterweil, and Brian Dickson. *RFC 7908: Problem Definition and Classification of BGP Route Leaks*. 2016. URL: *https://www.rfc-editor.org/info/rfc7908*.

[19] Andree Toonk. *Massive route leak causes internet slowdown*. Accessed: 2025-05-20. 2015. URL: *https://www.bgpmon.net/massive-route-leak-cause-internet-slowdown/*.

[20] Geoff Huston. *Analysis of a Route Leak*. Accessed: 2025-05-20. 2025. URL: *https://labs.apnic.net/index.php/2025/05/05/analysis-of-a-route-leak/*.

[21] Tom Strickx. *How Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Today*. Accessed: 2025-05-21. 2019. URL: *https://blog.cloudflare.com/how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-today/*.

[22] Lorin Hochstein. *Quick takes on the recent OpenAI public incident write-up*. Accessed: 2025-05-21. 2024. URL: *https://surfingcomplexity.blog/2024/12/14/quick-takes-on-the-recent-openai-public-incident-write-up/*.

[23] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. “Probabilistic verification of network configurations”. In: *Proceedings of the ACM SIGCOMM 2020 Conference*. 2020.

[24] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. “Seamless network-wide IGP migrations”. In: *Proceedings of the ACM SIGCOMM 2011 Conference*. 2011.

[25] Stefano Vissicchio, Laurent Vanbever, Cristel Pelsser, Luca Cittadini, Pierre Francois, and Olivier Bonaventure. “Improving network agility with seamless BGP reconfigurations”. In: *IEEE/ACM Transactions on Networking* 21.3 (2012).

[26] Pierre Francois and Olivier Bonaventure. “Avoiding transient loops during the convergence of link-state routing protocols”. In: *IEEE/ACM Transactions on Networking* 15.6 (2007).

[27] Francois Clad, Stefano Vissicchio, Pascal Mérindol, Pierre Francois, and Jean-Jacques Pansiot. “Computing minimal update sequences for graceful router-wide reconfigurations”. In: *IEEE/ACM Transactions on Networking* 23.5 (2014).

[28] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. “Don’t Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations”. In: *Proceedings of the ACM SIGCOMM 2016 Conference*. 2016.

[29] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. “NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion”. In: *15th USENIX Symposium on Networked Systems Design and Implementation*. 2018.

[30] Kausik Subramanian, Loris D’Antoni, and Aditya Akella. “Synthesis of fault-tolerant distributed router configurations”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.1 (2018).

[31] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. “Practical intent-driven routing configuration synthesis”. In: *20th USENIX Symposium on Networked Systems Design and Implementation*. 2023.

[32] Tony Bates, Philip Smith, and Geoff Huston. *CIDR Report*. Accessed: 2025-05-23. 2025. URL: *https://www.cidr-report.org/as2.0/*.

[33] John Moy. *RFC 2328: OSPF Version 2*. Tech. rep. 1998. URL: *https://www.rfc-editor.org/info/rfc2328*.

[34] Dave Oran. *RFC 1142: OSI IS-IS Intra-domain Routing Protocol*. Tech. rep. 1990. URL: *https://www.rfc-editor.org/info/rfc1142*.

[35] Donnie Savage, James Ng, Steven Moore, Donald Slice, Peter Paluch, and Russ White. *RFC 7868: Cisco’s Enhanced Interior Gateway Routing Protocol (EIGRP)*. 2016. URL: *https://www.rfc-editor.org/info/rfc7868*.

[36] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008).

[37] Yakov Rekhter, Tony Li, and Susan Hares. *RFC 4271: A border gateway protocol 4 (BGP-4)*. Tech. rep. 2006. URL: *https://www.rfc-editor.org/info/rfc4271*.

[38] Lixin Gao and Jennifer Rexford. "Stable Internet Routing without Global Coordination". In: *IEEE/ACM Transactions On Networking* 9.6 (2001).

[39] Tony Bates, Enke Chen, and Ravi Chandra. *RFC 4456: Bgp route reflection: An alternative to full mesh internal bgp (ibgp).* Tech. rep. 2006. URL: *https://www.rfc-editor.org/info/rfc4456*.

[40] Benoît Claise. *RFC 3954: Cisco Systems NetFlow Services Export Version 9.* 2004. URL: *https://www.rfc-editor.org/info/rfc3954*.

[41] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. "Planck: Millisecond-scale monitoring and control for commodity networks". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014).

[42] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. "Packet-level telemetry in large datacenter networks". In: *Proceedings of the ACM SIGCOMM 2015 Conference.* 2015.

[43] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. "Stroboscope: Declarative network monitoring on a budget". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* 2018.

[44] Bruno Quoitin and Steve Uhlig. "Modeling the routing of an autonomous system with C-BGP". In: *IEEE network* 19.6 (2005).

[45] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. "A General Approach to Network Configuration Analysis". In: *12th USENIX Symposium on Networked Systems Design and Implementation.* 2015.

[46] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. "Formal methods for network performance analysis". In: *20th USENIX Symposium on Networked Systems Design and Implementation.* 2023.

[47] Divya Raghunathan, Maria Apostolaki, and Aarti Gupta. "A Layered Formal Methods Approach to Answering Queue-related Queries". In: *22th USENIX Symposium on Networked Systems Design and Implementation.* 2025.

[48] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *Proceedings of the ACM SIGCOMM 2017 Conference.* 2017.

[49] Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. "Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation". In: *Proceedings of the ACM SIGCOMM 2024 Conference.* 2024.

[50] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. "Toward formally verifying congestion control behavior". In: *Proceedings of the ACM SIGCOMM 2021 Conference.* 2021.

[51] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. "Groot: Proactive verification of dns configurations". In: *Proceedings of the ACM SIGCOMM 2020 Conference.* 2020.

[52] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. "NetKAT: Semantic foundations for networks". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

[53] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "Symnet: Scalable symbolic execution for modern networks". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016.

[54] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. "Probabilistic netkat". In: *Proceedings of the 25th European Symposium on Programming, ESOP 2016*. 2016.

[55] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. "KATch: A Fast Symbolic Verifier for NetKAT". In: *Proceedings of the 45st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2024.

[56] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. "Debugging the data plane with anteater". In: (2011).

[57] Peyman Kazemian, George Varghese, and Nick McKeown. "Header space analysis: Static checking for networks". In: *9th USENIX Symposium on Networked Systems Design and Implementation*. 2012.

[58] Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. "Modular Data Plane Verification for Compositional Networks". In: *Proceedings of the CoNEXT Student Workshop 2023*. 2023.

[59] Alex Horn, Ali Kheradmand, and Mukul Prasad. "Delta-net: Real-time network verification using atoms". In: *14th USENIX Symposium on Networked Systems Design and Implementation*. 2017.

[60] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. "P4v: Practical verification for programmable data planes". In: *Proceedings of the ACM SIGCOMM 2018 Conference*. 2018.

[61] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. "Fast control plane analysis using an abstract representation". In: *Proceedings of the ACM SIGCOMM 2016 Conference*. 2016.

[62] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. "Tiramisu: Fast multilayer network verification". In: *17th USENIX Symposium on Networked Systems Design and Implementation*. 2020.

[63] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. "Plankton: Scalable network configuration verification through model checking". In: *17th USENIX Symposium on Networked Systems Design and Implementation*. 2020.

[64] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. "Symbolic router execution". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022.

[65] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. "Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2016.

[66] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. "NV: An intermediate language for verification of network control planes". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2020.

[67] Divya Raghunathan, Ryan Beckett, Aarti Gupta, and David Walker. "ACORN Network Control Plane Abstraction using Route Nondeterminism". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2022.

[68] Nick Giannarakis, Alexandra Silva, and David Walker. "ProbNV: probabilistic verification of network control planes". In: *Proceedings of the 42st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2021.

[69] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. "Detecting network load violations for distributed control planes". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2020.

[70] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, et al. "A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures". In: *Proceedings of the ACM SIGCOMM 2024 Conference.* 2024.

[71] Behnaz Arzani, Sina Taheri, Pooria Namyar, Ryan Beckett, Siva Kesava Reddy Kakarla, and Elnaz Jalilipour. *Raha, A General Tool to Analyze WAN Degradation.* 2025.

[72] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "Control plane compression". In: *Proceedings of the ACM SIGCOMM 2018 Conference.* 2018.

[73] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. "Lightyear: Using modularity to scale bgp control plane verification". In: *Proceedings of the ACM SIGCOMM 2023 Conference.* 2023.

[74] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. "Modular control plane verification via temporal invariants". In: *Proceedings of the 44st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2023.

[75] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. "Kirigami, the verifiable art of network cutting". In: *IEEE/ACM Transactions on Networking* 32.3 (2024).

[76] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. "Robotron: Top-down network management at facebook scale". In: *Proceedings of the ACM SIGCOMM 2016 Conference.* 2016.

[77] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. "Safely and automatically updating in-network ACL configurations with intent language". In: *Proceedings of the ACM SIGCOMM 2019 Conference*. 2019.

[78] Aris Leivadeas and Matthias Falkner. "A survey on intent-based networking". In: *IEEE Communications Surveys & Tutorials* 25.1 (2022).

[79] Ning Wang, Kin Hon Ho, George Pavlou, and Michael Howarth. "An overview of routing optimization for internet traffic engineering". In: *IEEE Communications Surveys & Tutorials* 10.1 (2008).

[80] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "B4: Experience with a Globally Deployed Software Defined WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference*. 2013.

[81] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference*. 2013.

[82] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. "TEAVAR: striking the right utilization-availability balance in WAN traffic engineering". In: *Proceedings of the ACM SIGCOMM 2019 Conference*. 2019.

[83] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. "Automatically repairing network control planes using an abstract representation". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, 359.

[84] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "Network-Wide Configuration Synthesis". In: *Computer Aided Verification*. 2017.

[85] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. "AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations". In: *Proceedings of the 16th ACM International Conference on Emerging Networking Experiments and Technologies*. 2020.

[86] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[87] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. "Network Configuration Synthesis with Abstract Topologies". In: *Proceedings of the 38st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.

[88] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. "What do LLMs need to synthesize correct router configurations?" In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023.

[89] Zhenbei Guo, Fuliang Li, Jiaxing Shen, Tangzheng Xie, Shan Jiang, and Xingwei Wang. "ConfigReco: Network configuration recommendation with graph neural networks". In: *IEEE Network* 38.1 (2023).

[90] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. "NetConfEval: Can LLMs Facilitate Network Configuration?" In: *Proceedings of the 20th ACM International Conference on Emerging Networking Experiments and Technologies*. 2024.

[91] Fuliang Li, Haozhi Lang, Jiajie Zhang, Jiaxing Shen, and Xingwei Wang. *PreConfig: A Pretrained Model for Automating Network Configuration*. 2024. URL: *https://arxiv.org/abs/2403.09369*.

[92] Jianmin Liu, Li Chen, Dan Li, and Yukai Miao. "CEGS: Configuration Example Generalizing Synthesizer". In: *22th USENIX Symposium on Networked Systems Design and Implementation*. 2025.

[93] Gonzalo Gomez Herrero and Jan Antón Bernal Van der Ven. *Network Mergers and Migrations: Junos Design and Implementation*. John Wiley & Sons, 2011.

[94] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. "Consistent updates for software-defined networks: Change you can believe in!" In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. 2011.

[95] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. "Abstractions for network update". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012).

[96] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. "Survey of consistent software-defined network updates". In: *IEEE Communications Surveys & Tutorials* 21.2 (2018).

[97] Pavol Černỳ, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. "Optimal consistent network updates in polynomial time". In: *Distributed Computing: 30th International Symposium*. 2016.

[98] Kim G Larsen, Anders Mariegaard, Stefan Schmid, and Jiří Srba. "Allsynth: A bdd-based approach for network update synthesis". In: *Science of Computer Programming* 230 (2023).

[99] Kedar S Namjoshi, Sougol Gheissi, and Krishan Sabnani. "Algorithms for In-Place, Consistent Network Update". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024.

[100] Ratul Mahajan and Roger Wattenhofer. "On consistent updates in software defined networks". In: *Proceedings of the 12th ACM Workshop on Hot Topics in Networks*. 2013.

[101] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference*. 2013.

[102] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. "Efficient synthesis of network updates". In: *ACM Sigplan Notices* 50.6 (2015).

[103] Stefano Vissicchio and Luca Cittadini. "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates". In: *Proceedings of the IEEE 35th International Conference on Computer Communications (INFOCOM)*. 2016.

[104] Richard Alimi, Ye Wang, and Y. Richard Yang. "Shadow Configuration as a Network Management Primitive". In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008).

[105] Zibin Chen and Lixin Gao. "CURSOR: Configuration update synthesis using order rules". In: *Proceedings of the IEEE 42th International Conference on Computer Communications (INFOCOM)*. 2023.

[106] João Luis Sobrinho. "Algebra and algorithms for QoS path computation and hop-by-hop routing in the Internet". In: *Proceedings of the IEEE 20th International Conference on Computer Communications (INFOCOM)*. 2001.

[107] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. "The stable paths problem and interdomain routing". In: *IEEE/ACM Transactions On Networking* 10.2 (2002).

[108] Joao Luis Sobrinho. "Network routing with path vector protocols: Theory and applications". In: *Proceedings of the ACM SIGCOMM 2003 Conference*. 2003.

[109] Joao L Sobrinho. "An algebraic theory of dynamic network routing". In: *IEEE/ACM Transactions on Networking* 13.5 (2005).

[110] Joao Luis Sobrinho and Miguel Alves Ferreira. "Routing on Multiple Optimality Criteria". In: *Proceedings of the ACM SIGCOMM 2020 Conference*. 2020.

[111] Joao Luis Sobrinho. "Correctness of Routing Vector Protocols as a Property of Network Cycles". In: *IEEE/ACM Transactions on Networking* 25.1 (2017).

[112] Steve Uhlig, Bruno Quoitin, Jean Lepropre, and Simon Balon. "Providing public intradomain traffic matrices to the research community". In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006).

[113] Justin Furuness, Cameron Morris, Reynaldo Morillo, Amir Herzberg, and Bing Wang. "Bgpy: The bgp python security simulator". In: *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*. 2023.

[114] Ege Cem Kirci, Valerio Torsiello, and Laurent Vanbever. "What is the next hop to more granular routing models?" In: *Proceedings of the 23th ACM Workshop on Hot Topics in Networks*. 2024.

[115] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. "The internet topology zoo". In: *IEEE Journal on Selected Areas in Communications* 29.9 (2011).

[116] Roland Schmid. "On Fault-Tolerance and Tolerated Failures in Communication Networks". PhD thesis. 2025.

[117] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. "Lessons from the evolution of the Batfish configuration analysis tool". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023.

[118] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. "Campion: debugging router configuration differences". In: *Proceedings of the ACM SIGCOMM 2021 Conference*. 2021.

[119] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

[120] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008).

[121] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. "Evolving SMT strategies". In: *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2016.

[122] Mislav Balunovic, Pavol Bielik, and Martin Vechev. "Learning to Solve SMT Formulas". In: *Advances in Neural Information Processing Systems*. 2018.

[123] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962).

[124] Luca Cittadini, Massimo Rimondini, Stefano Vissicchio, Matteo Corea, and Giuseppe Di Battista. "From theory to practice: Efficiently checking BGP configurations for guaranteed convergence". In: *IEEE Transactions on Network and Service Management* 8.4 (2011).

[125] Randy Bush, Timothy G Griffin, Jun Li, Zhuoqing M Mao, Eric Purpus, and Daniel Stutzbach. "Happy packets to you!" In: *Proceedings of the IEEE 24th International Conference on Computer Communications (INFOCOM)*. 2005.

[126] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. "A measurement study on the impact of routing events on end-to-end Internet path performance". In: *ACM SIGCOMM Computer Communication Review* 36.4 (2006).

[127] Pascal Merindol, Pierre David, Jean-Jacques Pansiot, Francois Clad, and Stefano Vissicchio. "A fine-grained multi-source measurement platform correlating routing transitions with packet losses". In: *Computer Communications* 129 (2018).

[128] Laurent Vanbever. "Methods and techniques for disruption-free network reconfiguration". PhD thesis. 2012.

[129] Jay Borkenhagen, Randy Bush, Ron Bonica, and Serpil Bayraktar. *RFC 8642: Policy Behavior for Well-Known BGP Communities*. Tech. rep. 2019. URL: *https://www.rfc-editor.org/info/rfc8642*.

[130] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. "DOTE: Rethinking (Predictive) WAN Traffic Engineering". In: *20th USENIX Symposium on Networked Systems Design and Implementation*. 2023.

[131] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, et al. "Reasoning about network traffic load property at production scale". In: *21st USENIX Symposium on Networked Systems Design and Implementation*. 2024.

[132] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. "Robust Validation of Network Designs under Uncertain Demands and Failures". In: *14th USENIX Symposium on Networked Systems Design and Implementation*. 2017.

[133] Fatai Zhang, Oscar Gonzalez de Dios, Matt Hartley, Zafar Ali, and Cyril Margaria. *RFC 8001: RSVP-TE Extensions for Collecting Shared Risk Link Group (SRLG) Information.* Tech. rep. 2017. URL: *https://www.rfc-editor.org/info/rfc8001*.

[134] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. "The case for separating routing from routers". In: *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture.* 2004.

[135] Timothy G Griffin and Gordon Wilfong. "On the correctness of IBGP configuration". In: *ACM SIGCOMM Computer Communication Review* 32.4 (2002).

[136] Virginie Van den Schrieck, Pierre Francois, and Olivier Bonaventure. "BGP add-paths: the scaling/performance tradeoffs". In: *IEEE Journal on Selected Areas in Communications* 28.8 (2010).

[137] Stefano Vissicchio, Luca Cittadini, Laurent Vanbever, and Olivier Bonaventure. "iBGP deceptions: More sessions, fewer routes". In: *Proceedings of the IEEE 31th International Conference on Computer Communications (INFOCOM).* 2012.

[138] Robert Raszuk, Bruno Decraene, Christian Cassar, Erik Aman, and Kevin Wang. *RFC 9107: BGP Optimal Route Reflection (BGP ORR).* Tech. rep. 2021. URL: *https://www.rfc-editor.org/info/rfc9107*.

[139] Jim Uttaro, Pierre Francois, Keyur Patel, Jeffrey Haas, Adam Simpson, and Roberto Fra gassi. *RFC Draft: Best Practices for Advertisement of Multiple Paths in IBGP.* Tech. rep. 2016.

[140] Bernard Fortz and Mikkel Thorup. "Internet traffic engineering by optimizing OSPF weights". In: *Proceedings of the IEEE 19th International Conference on Computer Communications (INFOCOM).* 2000.

[141] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Dr. Vijay Srinivasan, and George Swallow. *RFC 3209: RSVP-TE: Extensions to RSVP for LSP Tunnels.* Tech. rep. 2001. URL: *https://www.rfc-editor.org/info/rfc3209*.

[142] Geoff Huston. *Measuring BGP in 2023—have we reached peak ipv4?* APNIC Blog. 2024. URL: *https://blog.apnic.net/2024/01/09/measuring-bgp-in-2023-have-we-reached-peak-ipv4/*.

[143] Vineet Bharti, Pankaj Kankar, Lokesh Setia, Gonca Gürsun, Anukool Lakhina, and Mark Crovella. "Inferring invisible traffic". In: *Proceedings of the 6th ACM International on Conference on emerging Networking Experiments and Technologies.* 2010.

[144] Jakub Mikians, Amogh Dhamdhere, Constantine Dovrolis, Pere Barlet-Ros, and Josep Solé-Pareta. "Towards a statistical characterization of the interdomain traffic matrix". In: *NETWORKING 2012: 11th International IFIP TC 6 Networking Conference.* Springer. 2012, 111.

[145] Gottfried Köthe and Gottfried Köthe. *Topological vector spaces.* Springer, 1983.

[146] Wenjia Fang and Larry Peterson. "Inter-AS traffic patterns and their implications". In: *Seamless Interconnection for Universal Services. Global Telecommunications Conference.* 1999.

[147] Matthew Roughan. "Simplifying the synthesis of Internet traffic matrices". In: *ACM SIGCOMM Computer Communication Review* 35.5 (2005).

[148] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. "Deriving traffic demands for operational IP networks: Methodology and experience". In: *IEEE/ACM Transactions On Networking* 9.3 (2001).

[149] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. "MATE: MPLS adaptive traffic engineering". In: *Proceedings of the IEEE 20th International Conference on Computer Communications (INFOCOM)*. 2001.

[150] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. "A roadmap for traffic engineering in SDN-OpenFlow networks". In: *Computer Networks* 71 (2014).

[151] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. "A survey on the contributions of software-defined networking to traffic engineering". In: *IEEE Communications Surveys & Tutorials* 19.2 (2016).

[152] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. "On low-latency-capable topologies, and their impact on the design of intra-domain routing". In: *Proceedings of the ACM SIGCOMM 2018 Conference*. 2018.

[153] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. "Semi-oblivious traffic engineering: The road not taken". In: *15th USENIX Symposium on Networked Systems Design and Implementation*. 2018.

[154] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. "R3: resilient routing reconfiguration". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. 2010.

[155] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. "Traffic engineering with forward fault correction". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. 2014.

[156] Michael Markovitch, Sharad Agarwal, Rodrigo Fonseca, Ryan Beckett, Chuanji Zhang, Irena Atov, and Somesh Chaturmohta. "TIPSY: predicting where traffic will ingress a WAN". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022.

[157] Yu-Wei Eric Sung, Sanjay Rao, Subhabrata Sen, and Stephen Leggett. "Extracting network-wide correlated changes from longitudinal configuration data". In: *Passive and Active Network Measurement: 10th International Conference, PAM 2009, Seoul, Korea, April 1-3, 2009. Proceedings 10*. 2009.

[158] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. "The evolution of network configuration: a tale of two campuses". In: *Proceedings of the 2011 Internet Measurement Conference*. 2011.

[159] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. "Management plane analytics". In: *Proceedings of the 2015 Internet Measurement Conference*. 2015.

[160] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. "Automatic life cycle management of network configurations". In: *Proceedings of the Afternoon Workshop on Self-Driving Networks*. 2018.

[161] Ann Bednarz. *Global Microsoft cloud-service outage traced to rapid BGP router updates*. 2023. URL: *https://www.networkworld.com/article/3686531/global-microsoft-cloud-service-outage-traced-to-rapid-bgp-router-updates.html* (visited on 02/13/2022).

[162] William R Parkhurst. *Cisco BGP-4 command and configuration handbook*. 2969. Cisco Press, 2001.

[163] Igor Griva, Stephen G Nash, and Ariela Sofer. *Linear and nonlinear optimization*. Vol. 108. Siam, 2009.

[164] John Forrest, Ted Ralphs, and Haroldo Gambini Santos. *coin-or/Cbc 2.10.8*. Version 2.10.8.

[165] Cisco. *Understand Route Aggregation in BGP*. Technical Report. Accessed: June 2023. 2013.

[166] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig. "Interdomain traffic engineering with BGP". In: *IEEE Communications Magazine* 41.5 (2003).

[167] Stefano Vissicchio, Luca Cittadini, and Giuseppe Di Battista. "On IBGP Routing Policies". In: *IEEE/ACM Transactions on Networking* 23.1 (2015).

[168] Pierre Francois, Mike Shand, and Olivier Bonaventure. "Disruption free topology reconfiguration in OSPF networks". In: *Proceedings of the IEEE 26th International Conference on Computer Communications (INFOCOM)*. 2007.

[169] Nikola Gvozdiev, Brad Karp, Mark Handley, et al. "LOUP: The Principles and Practice of Intra-Domain Route Dissemination". In: *10th USENIX Symposium on Networked Systems Design and Implementation*. 2013.

[170] Bernard Fortz and Mikkel Thorup. "Internet traffic engineering by optimizing OSPF weights". In: *Proceedings of the IEEE 19th International Conference on Computer Communications (INFOCOM)*. 2000.

[171] Walid Ben-Ameur and Eric Gourdin. "Internet routing and related topology issues". In: *SIAM Journal on Discrete Mathematics* 17.1 (2003).

[172] Paul Traina, Danny McPherson, and John Scudder. *RFC 5065: Autonomous system confederations for BGP*. Tech. rep. 2007. URL: *https://www.rfc-editor.org/info/rfc5065*.

[173] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. "Running BGP in Data Centers at Scale." In: *18th USENIX Symposium on Networked Systems Design and Implementation*. 2021.

[174] Sudipto Das, Omer Egecioglu, and Amr El Abbadi. "Anónimos: An LP-Based Approach for Anonymizing Weighted Social Network Graphs". In: *IEEE Transactions on Knowledge and Data Engineering* 24.4 (2012).

[175] Pravin M Vaidya. "Speeding-up linear programming using fast matrix multiplication". In: *30th annual symposium on foundations of computer science*. IEEE Computer Society. 1989.

[176] Andreas Bley. “Inapproximability results for the inverse shortest paths problem with integer lengths and unique shortest paths”. In: *Networks: An International Journal* 50.1 (2007).

[177] Mikael Call and Kaj Holmberg. “Complexity of inverse shortest path routing”. In: *International Conference on Network Optimization*. Springer. 2011.

[178] Michael R Garey and David S Johnson. *Computers and intractability: a Guide to the Theory of NP-Completeness*. Vol. 174. freeman San Francisco, 1979.

[179] Marco Chiesa, Luca Cittadini, Giuseppe Di Battista, Laurent Vanbever, and Stefano Vissicchio. “Using routers to build logic circuits: How powerful is BGP?” In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 2013.

[180] Stefano Vissicchio, Luca Cittadini, Laurent Vanbever, and Olivier Bonaventure. “iBGP deceptions: More sessions, fewer routes”. In: *Proceedings of the IEEE 31th International Conference on Computer Communications (INFOCOM)*. 2012.

[181] Chiara Orsini, Alistair King, Danilo Giordano, Vasileios Giotsas, and Alberto Dainotti. “BGPStream: a software framework for live and historical BGP data analysis”. In: *Proceedings of the 2016 Internet Measurement Conference*. 2016, 429.

[182] Thomas Alfroy, Thomas Holterbach, Thomas Krenc, KC Claffy, and Cristel Pelsser. “The Next Generation of BGP Data Collection Platforms”. In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024.

[183] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. “Software verification with VeriFast: Industrial case studies”. In: *Science of Computer Programming* 82 (2014).

[184] Paul Lipscombe. *Meta’s Facebook and Instagram suffer outage, more than 500,000 report issues*. Accessed: 2025-06-04. 2024. URL: *https://www.datacenterdynamics.com/en/news/metas-facebook-and-instagram-suffer-outage-more-than-200000-report-issues/*.

[185] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. “xBGP: When you can’t wait for the ietf and vendors”. In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020.

[186] Thomas Wirtgen, Tom Rousseaux, Quentin De Coninck, Nicolas Rybowski, Randy Bush, Laurent Vanbever, Axel Legay, and Olivier Bonaventure. “xBGP: Faster Innovation in Routing Protocols”. In: *20th USENIX Symposium on Networked Systems Design and Implementation*. 2023.