Towards Practical Network Verification and Repair in Real-World Networks

by

Ryusei Shiiba

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences

The Graduate University for Advanced Studies, SOKENDAI

March 2026

THESIS COMMITTEE

Thesis Supervisor

Dr. Kensuke Fukuda
Professor of the Graduate University for Advanced Studies (SOKENDAI)/ National Institute of Informatics

Thesis Readers

Dr. Yusheng Ji
Professor of the Graduate University for Advanced Studies (SOKENDAI)/ National Institute of Informatics

Dr. Takashi Kurimoto
Professor of the Graduate University for Advanced Studies (SOKENDAI)/ National Institute of Informatics

Dr. Ichiro Hasuo
Professor of the Graduate University for Advanced Studies (SOKENDAI)/ National Institute of Informatics

Dr. Takeru Inoue
Associate Professor of University of Yamanashi

Towards Practical Network Verification and Repair in Real-World Networks

by

Ryusei Shiiba

Submitted to the Department of Informatics
on March, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Computer networks, such as the Internet, serve as a critical infrastructure for modern society. As the number of users and services continues to grow, computer networks become larger and more complex, making it difficult for operators to manage them without failures. Misconfigurations of network devices (e.g., routers and switches) remain a primary cause of network failures, often resulting in significant downtime. To address this, researchers have explored network verification and configuration repair approaches. Network verification automatically checks whether given configurations satisfy high-level policies of operators, whereas configuration repair synthesizes configuration changes (i.e., patches to configuration parameters) to enforce those policies. Despite their promise, these techniques are not yet widely adopted in real-world operations.

This dissertation advances the practicality of network verification and configuration repair by addressing two key challenges that hinder real-world deployment. The first challenge is the difficulty of defining complete and correct specifications. To fully leverage these techniques, network operators must precisely describe forwarding behaviors (e.g., packet header constraints and forwarding paths) and network properties (e.g., reachability and waypoint traversal) as specifications for verification and repair. However, formalizing such specifications is increasingly difficult due to the inherent complexity of modern networks. To mitigate this burden, we propose two new approaches. The first approach reduces the operational burden of writing specifications for repair. Specifically, this approach (1) identifies unintended changes in forwarding behaviors caused by initial (incomplete) specifications and (2) iteratively refines these specifications to prevent such changes. This iterative process ensures the safety of automated repairs and guides operators toward complete specifications with minimal manual intervention. The second approach reduces the burden for data plane verification. This approach enables operators to symbolically analyze packet forwarding behaviors using only partial packet headers or forwarding paths. We develop a query language and scalable algorithms to extract forwarding behaviors matching given queries. By allowing for symbolic data plane analysis with partial header and/or path information, this approach leverages data plane verification to identify the causes and impacts of network failures without requiring comprehensive specifications.

The second problem is the lack of expressiveness in modeling recent sophisticated network functions (e.g., traffic engineering and service function chaining). Both verification and

repair require precisely modeling packet forwarding behaviors with these functions. However, existing works often fail to model them with sufficient expressiveness while maintaining the scalability required for today's large-scale networks. To tackle this problem, we propose a new real-time data plane verification that simultaneously achieves high expressiveness and scalability. We develop a new data plane model to express sophisticated functions with formal network semantics and develop scalable verification algorithms on the model. This model and its algorithms enable operators to verify whether forwarding behaviors using these functions satisfy their intents in large-scale networks.

In all the works, we demonstrate the practical evaluation with real-world datasets (e.g., public WANs/LANs, production datacenter networks, and a real campus network), which show the scalability and effectiveness of our proposed approaches. Through these contributions, this dissertation makes the automated verification and repair techniques more accessible and reliable for real-world environments.

Acknowledgments

I would like to express my sincere gratitude to all those who supported me throughout my graduate studies. I could not have completed my Ph.D. program without my supervisor, Kensuke Fukuda. He taught me the fundamentals of research, from how to advance a project to how to write papers. Whenever I encountered a new research problem, he generously dedicated his time—even despite his demanding schedule—to engage in deep discussions with me. He also consistently provided invaluable support in my day-to-day life, including mental encouragement and securing the necessary funding. I am sincerely grateful for his unwavering support over the past five years. I am grateful for the advice from my advisors: Yusheng Ji, Takashi Kurimoto, Ichiro Hasuo, Megumi Kaneko, and Takeru Inoue. Their feedback and comments significantly improved my research and presentations. In particular, the intermediate evaluation provided a valuable opportunity to broaden my perspective, and I am grateful for their thoughtful insights. I am also greatly thankful to Satoru Kobayashi. Discussions with him consistently provided me with invaluable perspectives, particularly regarding the practical implications of my research for network operators. His constructive and timely comments on my papers were also invaluable in improving the quality of my work. In addition, he also provided invaluable support for my personal life, including assistance in finding accommodation and settling into the new environment. I am deeply grateful that he took time from his busy schedules to help my research and life. I am also greatly thankful to Osamu Akashi. Despite his busy schedule, he shared invaluable insights based on his extensive experience in network operations. His comments and feedback on both my research and the papers, from the perspectives of a network researcher and operator, greatly strengthened the practical relevance of this work. I would like to give special thanks to Ryo Nakamura and Yohei Kuga. Since my undergraduate years, they have taught me a wide range of knowledge, practical skills, and research in networked systems. In addition, they have guided me toward important research problems. Throughout my Ph.D. program, they also supported my experiments in real-world networks and provided valuable advice on writing papers. I am truly grateful for their long-standing support and guidance. I am also greatly thankful to Kenjiro Cho. He provided insightful and precise comments on my papers, which substantially improved their quality. He also supported me in many ways for my first attendance at SIGCOMM—from travel arrangements to guidance throughout the conference—which made the experience both smooth and memorable. I would like to express my sincere gratitude to Hiroki Shirokura at LY Corporation. His support enabled me to conduct experiments using realistic network datasets, which significantly strengthened this work. I am deeply grateful for his generous cooperation. Furthermore, his dedication to researching and developing networked systems has been a constant source of inspiration,

deepening my passion for the field. I feel truly fortunate to have had the opportunity to collaborate with him. I am also grateful for the interaction with all members of the CREST project, "Zero Trust IoT by Formal Verification and System Software." I am especially grateful to the formal verification specialists: Taro Sekiyama, Shin-ya Katsumata, Atsushi Igarashi, Kohei Suenaga, and Masaki Waga. I have greatly enjoyed discussing my research and formal verification with them. Our conversations and interactions have continually provided me with fresh insights and have been a tremendous source of motivation. I would like to thank all my colleagues in the Fukuda Laboratory for their support and encouragement. Our everyday conversations and discussions not only helped me throughout my research but also gave me many moments of relief and relaxation during my graduate studies. I am grateful to my undergraduate advisors at Keio University: Shigeya Suzuki, Achmad Husni Thamrin, Ryousuke Abe, Keiko Shigeta, Noriatu Kudo, Keiko Okawa, Osamu Nakamura, and Jun Murai. Their support gave me the confidence to pursue this Ph.D. program. I also thank all the collaborators in the WIDE Project, particularly Hirochika Asai, Hiroki Watanabe, Korry Luke, and Atsuya Osaki. Finally, I would like to thank my family for their unwavering understanding and support. Their encouragement and guidance have allowed me to focus on my research throughout my graduate studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overview of the dissertation. We first introduce the background of network infrastructures and failures (§1.1 and §1.2). We then discuss current network verification and repair approaches to prevent such failures, highlighting their limitations (§1.4 and §1.5). Finally, we present our three contributions to address these problems (§1.6).

1.1 Today's computer network infrastructure

Computer networks (e.g., the Internet) serve as a critical infrastructure for information systems and services in our daily lives. Various systems, including government, social networking, and video streaming, rely on stable network communication. As the dependency on these services grows, network stability becomes increasingly crucial.

The expansion of users and services has significantly increased network scale. Worldwide Internet users exceed 5.56 billion [1], and annual traffic grew by 17.2% in 2024 [2]. To accommodate this demand, operators continuously deploy network devices such as routers and switches. Consequently, modern networks, including data centers and mobile networks, now consist of thousands or tens of thousands of devices [3].

The increasing scale of computer networks has increased operational complexity. The large-scale networks are usually managed by over 10^5 to 10^6 lines of configurations and 10^6 Forwarding Information Base (FIB) entries per network device [4]. Furthermore, the networks are composed of not only traditional routing protocols (e.g., BGP and OSPF) but also fine-grained access control, traffic engineering, and sometimes SDN-like controls [5–7] on network devices of multiple vendors [8,9]. This sheer volume and variety of configurations and protocols make it difficult for network operators to fully comprehend and safely manage their networks.

In addition to the complexity, network operators in large-scale networks need to manage the various application demands on a single physical network [10,11]. Typically, each application has its own routing/forwarding demands, such as requiring specific security appliances or per-user and per-application isolation. Furthermore, these applications often impose strict performance requirements, demanding high throughput, low latency, and reliable packet delivery. To meet these diverse demands, operators employ recent sophisticated network techniques such as Traffic Engineering (TE), Service Function Chaining (SFC), and Virtual

Table 1.1: Typical elements in router configurations

Element	Example
Hardware settings	Interface speed, duplex, and MTU
Protocol settings for external route	Parameters for external routing (e.g., routing policy in eBGP)
Protocol settings for internal route	Parameters for internal routing (e.g., link cost in OSPF)

Private Network (VPN) that customize packet forwarding behavior beyond shortest-path routing [12–18]. While these techniques provide flexibility, they introduce a new layer of complexity: operators must now manage application-specific behaviors on top of the underlying network connectivity.

1.2 Network failures and a primary cause

With the increasing network scale and complexity, we have observed that data communication on the networks becomes unstable or is totally disconnected (i.e., network failures) [2,4,9,19–22]. A 2024 report from Cloudflare documented 225 major network outages worldwide [2]. These failures have severe real-world consequences, disrupting critical services ranging from emergency call systems [23] to enterprise email [24]. For instance, a BGP misconfiguration in Google's network led to the leak of approximately 160,000 prefixes. This failure makes traffic from major Japanese telecommunication providers unseen (called blackhole), including NTT OCN and KDDI, blocking access to online services for dozens of companies for several hours [20].

Network failures are not limited to large-scale incidents between global companies; they also frequently occur within the infrastructure of a single organization, such as in datacenter networks [8,19] and Wide Area Networks (WANs) [9,25]. These failures manifest in various forms, including missing forwarding entries, traffic black holes, and forwarding loops [9,26].

A primary cause of network failures is router misconfiguration by network operators in daily operation [8,9,19,22,25]. For example, 30% of network failures at Microsoft and 50% of network failures at Alibaba are due to router misconfigurations [8,22]. Router configurations consist of low-level parameters for routing protocols and hardware settings (examples in Table 1.1) that define how the routers communicate with each other to determine packet forwarding behaviors on the networks. Operators change the configurations to implement new routing/forwarding policies and change existing policies. For instance, operators in Meta's backbone networks change an average of 12.5 configurations per network device per week [27]. Each router run distributed route computation based on the configuration changes and routing protocols (e.g., BGP, OSPF, and ISIS). Then the resulting convergent state determines how the network forwards packets. There are several reasons why the misconfigurations happen at this stage. We focus on discussing the two major reasons in the following.

1. Semantic gap: There is a fundamental semantic gap between an operator's policy and a router's configuration. Operators plan configuration changes based on their high-level

operational policies for how packets should be forwarded on networks. In contrast, the router configuration formats require them to directly embed specific values into low-level parameters tied to routing protocol semantics and hardware-specific features. This translation process creates misconfigurations, which sometimes lead to severe network failures.

2. Scale and complexity: As discussed, the scale and complexity of router configurations in today's networks have grown beyond the capability of operators to correctly reason. Today's large-scale networks house over 10^3 to 10^4 of network devices and are managed by over 10^5 to 10^6 lines of configurations per device. In addition, each configuration contains several components to meet the routing/forwarding demands of applications and services. Due to the scale and complexity, it is a tremendous task for operators to understand all the existing configurations and make new changes without any mistakes.

1.3 Common Practices to Preventing Network Failures

In real-world operations, network operators employ a range of practices to prevent network failures and to minimize the impacts when the failures occur. Because the failures often stem from subtle interactions among routing policies, configurations of network devices, and dynamic network conditions (e.g., link status and utilization rate), operators typically combine (i) runtime observation of the networks (i.e., network testing and monitoring) and (ii) pre-deployment checks of planned configuration changes.

A widely adopted and operationally mature approach is network testing and monitoring. Operators use active probing tools (e.g., ping and traceroute) and telemetry to continuously validate end-to-end connectivity and performance between endpoint hosts. The outputs from these testing and monitoring reflects the quality of service experienced by the endpoint hosts, which are helpful to detect network failures and their impacts. In addition, the end-to-end approach can detect gray failures that may not be visible by observing the current status of each network device individually.

Network operators also take proactive measures before deploying configuration changes. The most common practice is manual validation, including peer review, change checklists, and staged rollouts. Furthermore, most operators employ lightweight automated checks, such as configuration linting, schema validation, and unit-test-like assertions for common policies (e.g., ensuring that critical prefixes are not filtered or that required sessions are configured). These checks are effective for catching obvious mistakes and enforcing local invariants, and they fit naturally into existing operational workflows.

However, these testing/monitoring and proactive checks are often limited in scope. As networks and policies grow in size and complexity, it becomes increasingly difficult to comprehensively test and monitor all traffic classes among all network devices and instances. Manual review does not scale well and may miss faulty end-to-end behaviors that arise only from the interaction of multiple routers and protocols. Similarly, lightweight automated checks tend to validate syntactic correctness or local policies rather than end-to-end network-wide behaviors. These limitations motivate more comprehensive approaches that can reason about all possible network-wide behaviors from configurations and topology, and can provide stronger correctness guarantees.

Table 1.2: Examples of network properties to be verified.

Property	**Description**
Reachability	Node (group) A reaches node (group) B
Waypoint	Node (group) A reaches node (group) B through node (group) C
Isolation	Node (group) A can not reach node (group) B
Hop-limit	Node (group) A reaches node (group) B within n hop
Hop-consistency	Node group A reaches node group B with n hop

1.4 Automatic configuration repair and verification

Motivated by the limitations of testing/monitoring and lightweight pre-deployment checks, prior work has investigated *network verification* and *automatic configuration repair* as more comprehensive approaches to preventing network failures induced by configuration changes. Both approaches *symbolically* model route convergence processes in networks and compute the resulting packet forwarding behaviors as pairs of packet header constraints and their forwarding paths. This symbolic computation enables us to (1) consider all possible routing/forwarding behaviors and (2) check which types of network properties are satisfied with given router configurations, packet forwarding rules, and network topology (details in chapter 2).

Configuration repair: Configuration repair aims to safely automate the task of configuration changes by finding new parameters in configurations to satisfy high-level policies of network operators [28–31]. These routing and forwarding policies are usually expressed as forwarding paths based on a regular expression with path preference (called specifications). Given router configurations, network topology, and specifications, this approach (1) models route propagation processes and (2) generates patches indicating how to fix the configurations to satisfy the specifications. To improve scalability for real-world deployment, recent tools incorporate symbolic reasoning and/or heuristic search in both modeling and patch generation [29–31]. Overall, configuration repair offers a promising path toward automating configuration changes while reducing the risk of misconfigurations.

Control and Data plane verification: In contrast to the repair approach, control and data plane verification aim to automatically *verify* whether operator's policies are satisfied or not in networks. These policies are written by a pair of forwarding path requirements (as with configuration repair) and network properties in Table 1.2. Control plane verification takes router configurations as input and checks whether the converged routing states satisfy the given policies [32–40]. On the other hand, data plane verification takes packet forwarding rules (e.g., FIBs and ACLs) generated after route computation processes as input and checks whether the current forwarding behaviors satisfy the given policies [26,41–52]. Accordingly, control plane verification is well suited for pre-deployment validation of configurations, whereas data plane verification validates the forwarding behaviors implemented by the current data plane states.

1.5 Problem of existing repair and verification

While some big cloud companies use the repair and verification techniques in their daily operation [9,25,33,53], these techniques have yet to see widespread adoption in practice [33,54]. Among the several reasons for this slow adoption, this dissertation focuses on the following two key issues.

1.5.1 Difficulty of writing correct specification

Both the repair and verification techniques require operators to write specifications describing how networks forward packets. Concretely, a specification consists of (1) packet header constraints and (2) their forwarding paths. Given a set of specifications, the repair technique changes given configurations such that the convergent routing state satisfies them. The verification technique checks whether the convergent state with given configurations satisfies the specifications.

However, writing specifications that completely capture all of the intended policies for repair and verification is tremendously difficult in practice. The primary reason is that the complexity of the required specifications grows in direct proportion to the scale and complexity of the network itself. For instance, operators in a global backbone maintain 10^6 classes of traffic with distinct forwarding paths in their daily operation [55]. Understanding all the traffic and writing the complete specifications that manage the traffic is practically impossible. Specification mining techniques could ease the difficulties by automatically enumerating all the existing policies [56,57]. However, how to understand and manage the enumerated (huge) specifications is an open problem.

If operators do not write important routing/forwarding policies in the specifications, neither repair nor verification can work as intended. For repair, the incomplete specifications could introduce unexpected changes to existing routing behaviors (details in chapter 3). For verification, operators can not check whether their intended properties are satisfied in networks (details in chapter 4).

1.5.2 Model expressiveness for sophisticated network functions

To repair and verify router configurations, we precisely model packet forwarding behaviors in today's networks. As described in §1.1, operators in large-scale networks use recent sophisticated network techniques such as Traffic Engineering (TE) and Service Function Chaining (SFC) to meet the routing/forwarding demands of applications and services to networks [12–18]. Therefore, we need network models that have enough expressiveness to model such network techniques.

However, most prior works can not fully express header transformations (packet encapsulation, decapsulation, and rewrite) on their models [26,41,43,45,47,48,50,58]. These transformations are primitive packet processing to model the sophisticated techniques. Because of the limited expressiveness, the prior works are not suitable for verifying these techniques. APT [42] and Katra [44] can handle these three header transformations; however, they have non-negligible overhead for verifying the large-scale networks. Therefore, how

to model the sophisticated techniques in a reasonable time for daily operation is an open problem.

1.6 Contributions

This dissertation aims to improve the practical applicability of network repair and verification techniques by addressing the two important problems (§1.5.1 and §1.5.2). Figure 1.1 illustrates the overview of the contributions. We address the problem in §1.5.1 for configuration repair and data plane verification. For the problem in §1.5.2, we improve the expressiveness of existing data plane models, which can be applicable for network verification and configuration repair. We summarize each contribution as follows.

1. Iterative Specification Refinement for Configuration Repair (chapter 3): Configuration changes on network devices are a routine task for network operators; however, misconfigurations sometimes cause significant downtime. Configuration repair aims to safely automate this task by inferring new configuration parameters to satisfy operator's high-level network change intents. Despite its benefit, existing repair tools could introduce unexpected forwarding path changes not described by the change intents (i.e., side effects). These side effects hinder their real-world deployment, as such path changes occur without the operators' awareness. We propose ISR, a new augmentative framework for repair tools to prevent undesired side effects in satisfying network change intents. The key idea is iterative specification refinement: ISR (1) identifies side effects in configurations after repair, (2) generates minimal repair specifications to prevent them, and (3) feeds these specifications back into the next repair. This iterative process guides repair tools to find configuration parameters that satisfy change intents without introducing undesired side effects. We evaluate the effectiveness and scalability of ISR on real network topologies. We first demonstrate that ISR successfully guides three repair tools (SMT/Simulation/LLM-based) to generate configurations that satisfy change intents without undesired side effects in OSPF and BGP networks. Second, scalability evaluation shows that ISR completes a refinement iteration within seconds for varying numbers of side effects on the topologies.

2. Symbolic Extraction of Packet Forwarding Behavior with Partial Queries (chapter 4): While data plane verification can explore all possible behaviors, it is difficult for network operators to write correct specifications for daily operational usage. To ease the difficulty while achieving the benefit, we propose Lupe, a system that enables operators to (1) specify *symbolic* packet headers and/or forwarding paths and (2) obtain all packet forwarding behaviors matching the symbolic values for identifying the failure causes and impacts. Out of packet forwarding rules and network topology, Lupe automatically extracts forwarding behaviors matching given symbolic headers and forwarding paths. For instance, operators can ask the following questions for Lupe by single queries: (1) "Show forwarding paths destined to a suspicious prefix related to a failure" and (2) "Show all types of packet headers forwarded through a flapped link." We design efficient data models and algorithms for the extraction and a query language for operators to easily specify the queries. Our evaluation shows that Lupe extracts forwarding behaviors matching given symbolic headers or paths in a production datacenter network (>2000 network devices) in 15 seconds. Furthermore, we confirm the usefulness of Lupe through two case studies to help operators identify the causes and impacts

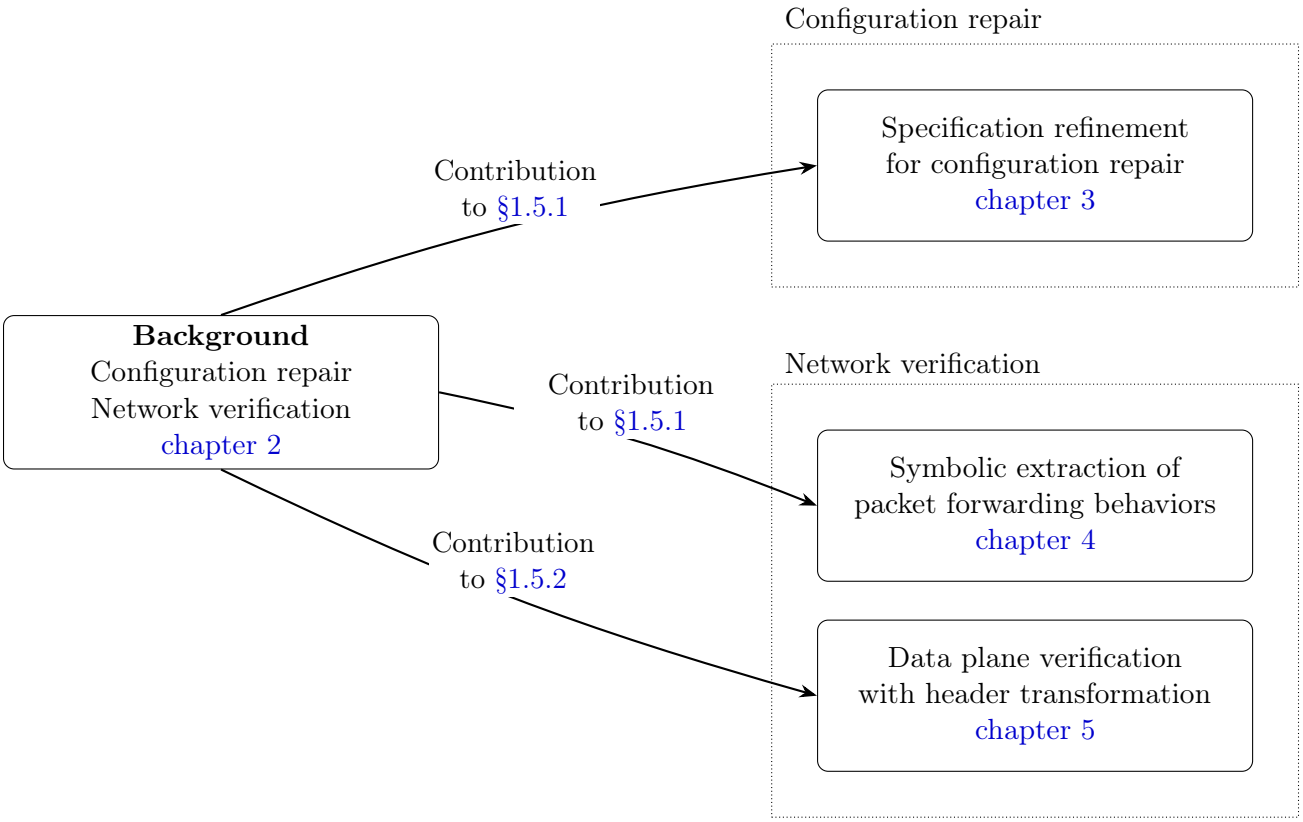


Figure 1.1: Overview of this dissertation and our contributions.

of real failures in campus and production datacenter networks.

3. Real-time Data Plane Verification with Header Transformation (chapter 5): We present Graft, a new real-time data plane verification framework that simultaneously achieves scalability for large-scale networks and model expressiveness for sophisticated network functions. For scalable realtime verification, we first propose an optimized algorithm to efficiently compute and manage large packet header spaces and their forwarding paths. Second, we propose a data plane model and algorithms with formal network semantics to precisely model forwarding behavior with sophisticated functions. We validate its effectiveness using synthetic and production datacenter networks. To the best of our knowledge, we are the first to verify the forwarding behavior in production large-scale networks. For scalability, we show that Graft is 100x faster than prior works in the synthetic networks and 20000x faster in the production network. For expressiveness, we demonstrate that Graft is enough to model the customized forwarding behavior by verifying the correctness of SRv6-based SFCs in the production network. Finally, we demonstrate that Graft verifies a real failure of a distributed NAT system in the production network.

1.7 Information for readers

Figure 1.1 guides the relationship among the chapters. First, we introduce network basics (control plane and data plane), network testing, verification, and repair approach in chapter 2.

Then we explain our three contributions from chapter 3 to chapter 5. Finally, we discuss the contributions in chapter 6.

Chapter 2

Background

This chapter provides an overview of fundamental network concepts, network verification, and configuration repair. §2.1 outlines routing and forwarding mechanisms, covering key protocols such as BGP and OSPF, as well as network functions like Network Address Translation (NAT). §2.2 introduces network testing and monitoring. Subsequently, §2.3 and §2.4 then discuss network verification. While this research area includes emerging approaches such as performance verification [59,60] and machine learning with formal methods [61], we focus on two key topics: control plane and data plane verification. Finally, §2.5 provides an overview of configuration repair.

2.1 Routing and forwarding

Computer networks are built by connecting network devices to each other and exchanging the routing information between routers. Figure 2.1 illustrates the architecture of a router. Router configurations define how a router announces and exchanges routes via routing protocols and specify hardware settings such as names and IP addresses of physical interfaces (ports). Given the configurations, the router runs instances of routing protocols and communicates with other routers. The routing protocols are generally classified into two types: Interior Gateway Protocols (IGP) and Border Gateway Protocol (BGP). We explain how the protocols work in §2.1.1 and §2.1.2. After the protocol processing, the router creates Routing Information Base (RIB), a database that manages all route information exchanged by the other routers (details in §2.1.1). Because not all learned routes are required for forwarding, the router generates a Forwarding Information Base (FIB), a table containing the subset of routes necessary for packet forwarding (details in §2.1.2). The router refers to the FIB to forward packets and the RIB to manage received route information.

2.1.1 Control plane protocol and routing

Each router in a network exchanges its own route information and computes how to route receiving traffic. The process of selecting a path across one or more networks is referred to as *routing*. Control plane protocols define how each router runs this path selection process, working together. The protocols are classified into (1) the protocol for inter-domain networks

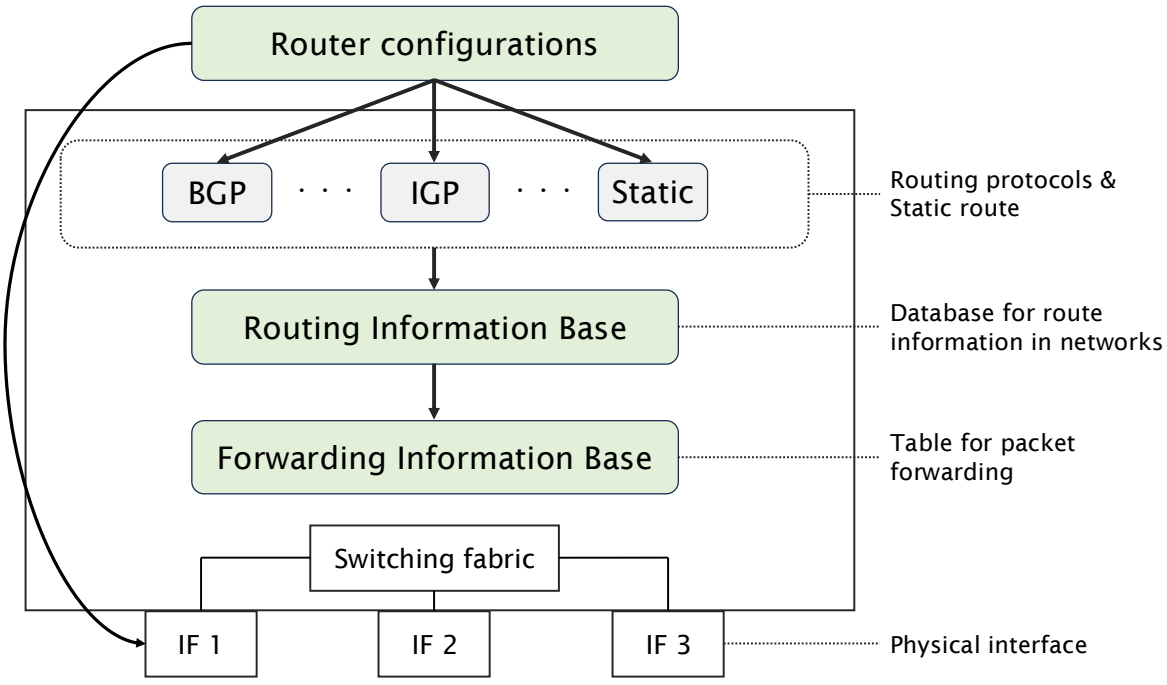


Figure 2.1: High-level illustration of a router architecture.

(i.e., BGP) and (2) the protocols for intra-domain networks (i.e., IGP).

BGP: An Autonomous System (AS) is a collection of IP networks managed by a single administrative entity. BGP, an exterior gateway protocol (EGP) that operates over TCP port 179 to ensure reliable delivery, is the standard protocol for exchanging routing information between ASes. As shown in Figure 2.2, an AS (e.g., AS 100) advertises IP prefixes—groups of IP addresses—to its neighbors, signaling its ability to provide reachability to them. BGP sessions are further categorized as External BGP (eBGP) between different ASes or Internal BGP (iBGP) within the same AS. When a BGP router receives route advertisements, it first applies an import policy. The router then processes these candidate routes through its best-path selection algorithm—evaluating attributes such as LOCAL_PREF, AS_PATH length, and the reachability of the NEXT_HOP—to determine the single route for each prefix. The best route is installed in the router's Routing Information Base (RIB) and is then considered for advertisement to other neighbors, subject to an export policy. This propagation of route information continues until the network's routing state converges, a process that can be more gradual than interior protocols due to the global scale of the Internet.

BGP is a path vector protocol. Each route advertisement includes the AS_PATH attribute, which records the sequence of ASes the route has traversed, providing a loop-prevention mechanism. In addition to AS_PATH, BGP advertisements contain several other attributes used in the best-path selection process. For instance, LOCAL_PREF attribute allows an AS to express a preference for a specific exit point; a higher value is preferred. In Figure 2.2, AS 400 prefers the path through AS 200 over the one through AS 300 because the route from AS 200 is configured with a higher local preference (300) than the default value (100) assigned to the route from AS 300. Network operators leverage these attributes to implement

sophisticated routing policies. For example, an operator might influence inbound traffic by using AS_PATH prepending to make a path less attractive, or filter advertisements based on business relationships, such as those with peers versus transit providers.

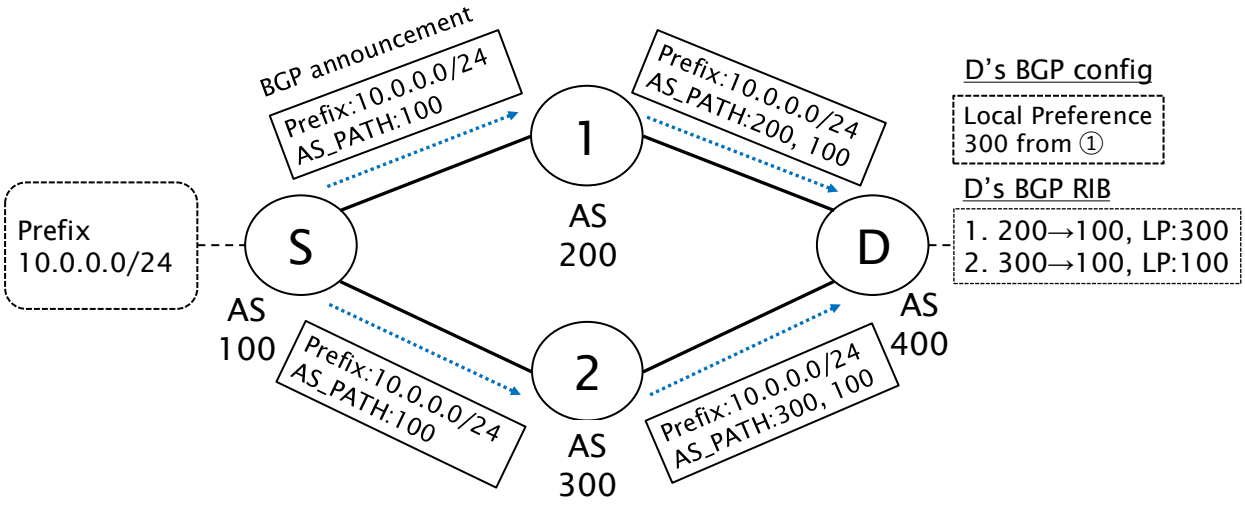


Figure 2.2: BGP route announcement example.

IGP: Interior Gateway Protocols (IGP) are routing protocols used to ensure connectivity between routers within a single AS. These protocols define a route computation process that facilitates the exchange of route information among routers in a single network.

One common protocol is Open Shortest Path First (OSPF), which is a link-state protocol used to find the best paths among routers based on a numerical metric (cost) typically derived from link bandwidth. Before exchanging routing data, OSPF routers use a "Hello" protocol to discover neighbors and establish adjacencies. For scalability, OSPF allows a network to be divided into areas, with all areas connecting to a central backbone area (Area 0). Using OSPF, each router builds a complete topological map of its network area. Routers flood Link-State Advertisements (LSAs), packets containing connected link information, to all other routers in the area. This process ensures that every router maintains an identical database of the network's topology called a link-state database (LSDB).

With this map, each router independently runs the Dijkstra algorithm (also known as the Shortest Path First algorithm) to calculate the shortest, loop-free path from itself to every destination, based on link costs, using itself as the root of the SPF tree. If there are multiple paths that have the same cost, the router performs Equal-Cost Multi-Path (ECMP), where traffic between the source and destination nodes of the paths is equally distributed. After the calculation, these shortest paths are stored in the OSPF RIB. This link-state nature provides OSPF with several key advantages, including fast convergence, as routers can perform triggered updates and recalculate paths immediately upon a topology change.

Figure 2.3 illustrates a simplified example of the shortest path computation at router Ⓓ. Each router calculates its own Shortest Path First Tree (SPF tree), with itself as the root, to determine the shortest path to all other routers. For instance, the minimal cost of the path from Ⓓ to Ⓢ is 15; therefore, router Ⓓ (router id 4.4.4.4) selects the path through router ① (router id 1.1.1.1) to reach Ⓢ.

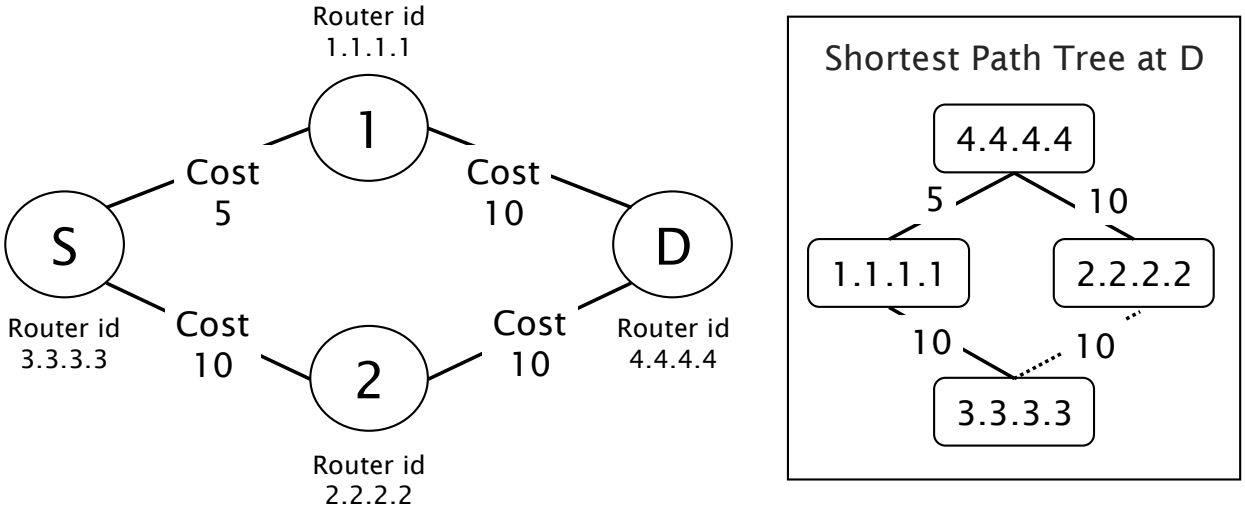


Figure 2.3: An example OSPF network and the shortest path tree at router Ⓓ. Unlike LSDB, which describes a complete topological map, the shortest path tree for each router only contains the shortest paths between the router and the others.

Static route: While dynamic routing protocols like BGP and OSPF automatically learn network paths, static routes are manually configured by network operators to direct traffic to specific destinations. A key advantage is its low Administrative Distance (typically 1), which gives it higher precedence over routes learned dynamically. This ensures that the traffic follows a strictly defined path regardless of protocol calculations. Furthermore, operators can implement Floating Static Routes by adjusting the priority, providing a reliable manual backup mechanism. Since it requires no protocol overhead, it is the most resource-efficient method for stable environments, though it lacks the scalability of dynamic routing in large-scale, complex topologies.

2.1.2 Data plane forwarding and functions

Based on the control plane's best path calculations, routers and switches forward incoming packets between network interfaces to maintain connectivity. This high-speed forwarding functionality is defined as the *Data plane*. While the Routing Information Base (RIB) contains the comprehensive set of available routes, it is not optimized for real-time forwarding. Consequently, the control plane derives a Forwarding Information Base (FIB), which is specifically structured for rapid lookup. The FIB contains only the active best paths for each destination prefix and their corresponding next-hop interfaces.

To achieve maximum performance, routers offload the FIB into specialized, high-speed memory—often Ternary Content Addressable Memory (TCAM)—located on the line cards. Packet forwarding is then executed by Application-Specific Integrated Circuits (ASICs). When a packet enters an input port, the ASIC performs a hardware-based lookup of the destination address against the FIB and switches the packet to the appropriate egress interface. This process occurs at wire speed, ensuring minimal latency without taxing the main CPU.

In addition to basic forwarding, the data plane performs four primitive operations: drop

(discarding packets due to security or congestion), rewrite (modifying headers, such as TTL decrement), encapsulate, and decapsulate (handling tunneling and Layer 2/3 transitions).
Drop: This function discards incoming packets without forwarding them to egress interfaces. Network operators use this function primarily through Access Control Lists (ACLs). An ACL acts as a hardware-enforced packet filter. When a packet arrives at an ingress interface, the data plane's specialized hardware (ASIC) inspects the packet's header. It evaluates the packet's information—such as source/destination IP addresses and transport-layer port numbers—against an ordered list of rules (Access Control Entries) in the ACL. Based on the first matching rule, the router makes an immediate decision to either permit (forward) or deny (drop) the packet.
Rewrite: This function involves modifying incoming packet headers, such as source and destination IP addresses. It is frequently used for services like Network Address Translation (NAT) in production environments. For instance, when a packet from a private IP address (e.g., 192.168.1.10) egresses a network, the router's ASIC performs a header rewrite. t (1) replaces the source IP with a public address and the source port with a unique temporary port, (2) records this mapping in a NAT translation table, and (3) recalculates the packet's IP and transport-layer checksums. Conversely, when a return packet reaches that public IP and temporary port from the outside, the router looks up the mapping in its translation table and performs a reverse rewrite, reverting the destination IP and port back to the original private ones and once again recalculating the checksums before forwarding the packet.
Encapsulate and Decapsulate: These functions involve adding or removing a new packet header (outer header) to or from incoming packets. This mechanism is commonly used in tunneling protocols (e.g., VPNs) to transport private data across a public network. For instance, when a host sends a packet with a private IP header from one site to another, a router (or a specialized network device) encapsulates the packet by prepending a new, public IP header. This outer header facilitates the routing of the encapsulated payload over the Internet. Upon arrival at the destination endpoint, the router receiving the packet decapsulates it by removing this outer header, which reveals the original packet to be forwarded to its final private destination.

2.2 Network testing and monitoring

2.2.1 Overview

Network testing and monitoring validate whether a network provides the required connectivity and performance (e.g., throughput and latency). Operators commonly rely on active probing tools (e.g., ping and traceroute) to assess properties such as reachability between end hosts. These tools measure end-to-end metrics—including availability, latency, and packet loss—rather than relying solely on the reported status of individual switches or routers. For example, operators measure round-trip time (RTT), which captures propagation and transmission delay between hosts, queuing delay at switches, and processing latency in the OS kernel network stack. Because these probes use standardized protocols (e.g., ICMP), the tools are largely vendor-independent. This independence is crucial for defining and tracking network service-level agreements (SLAs) across heterogeneous infrastructure. Besides, these tools are

mature and widely used in practice, requiring no explicit network model.

2.2.2 Existing Approaches

As modern networks grow in complexity and scale, traditional monitoring methods such as SNMP often fail to detect gray failures—problems such as packet loss, intermittent latency spikes, or blackholes in which devices appear operational but do not forward traffic correctly. Consequently, testing and monitoring in large-scale networks (e.g., data centers) often treat the network as a black box, with the primary goal of validating the quality of service actually experienced by end hosts. Prior research on network testing and monitoring is commonly categorized by the method of data collection: active probing and passive monitoring.

Active probing. Active probing is the most widely adopted approach for measuring reachability and latency [62–68]. It injects synthetic test packets into the network to observe forwarding behavior and to measure end-to-end performance metrics. By actively sending probes, operators can validate key network properties and detect gray failures that may not be apparent from passive traffic alone. Pingmesh is a representative system in this category, designed to measure latency between servers at massive scale [62]. It employs a hierarchical probing strategy (intra-rack, intra-pod, and inter-DC) and uses TCP SYN/ACK packets to mimic application connection behavior, so that probes traverse the same Equal-Cost Multi-Path (ECMP) routes as production traffic. NetNORAD builds on this idea but uses UDP probes with embedded timestamps [63]. This design supports faster detection cycles (seconds rather than minutes) and helps separate true network latency from delays introduced by the operating system kernel, yielding a cleaner estimate of network performance. NetBouncer advances failure localization using an IP-in-IP probing mechanism that enables servers to steer probes along specific network paths [66]. This capability helps pinpoint faulty links and devices by correlating successful and failed probes with the network topology, addressing the challenge of distinguishing device failures from link failures in complex data center networks.

Passive monitoring and telemetry. Passive approaches observe real user traffic and/or device state by collecting packet metadata or telemetry in real time [69–73]. One prominent direction is packet-level telemetry, exemplified by Everflow [72]. Everflow leverages the match-and-mirror capabilities of commodity switch ASICs to sample selected packets (e.g., those with TCP control flags) and mirror them to a central collector. This enables operators to reconstruct the path a packet traversed and to identify where (and potentially why) packets were dropped. Another line of work is inference-based fault localization. Systems such as 007 [69] monitor TCP statistics at end hosts to detect loss events. Rather than relying on switch telemetry—which can be misleading when devices report healthy status while silently dropping packets—007 uses a statistical voting mechanism: multiple end hosts correlate their observed loss patterns to pinpoint the faulty link or device, avoiding the need for specialized hardware support.

2.3 Control plane verification

2.3.1 Overview

Control plane verification aims to verify whether router configurations satisfy operator's policies [9,32,34–38,74–80]. We illustrate the overview of the verification process in Figure 2.4. The control plane verifier takes three inputs: (1) router configurations, (2) network topology, and (3) specifications. Each specification describes a forwarding-path requirement, defined as a pair of packet-header constraints and the expected forwarding behavior (i.e., the corresponding forwarding path). Given these inputs, the verifier checks the desired properties in the following three steps.

Preliminary. Control plane verification models networks with router configurations and its topology as a graph. Each node and edge in the graph corresponds to physical or virtual network devices and the links between them as illustrated in Figure 2.4.

1. Modeling route computation. First, control plane verifiers compute how each router send receiving traffic with the given configurations. To this end, the verifiers model and simulate the route computation processes of each routing protocols used in the configurations. After the simulation, they create a single routing table (i.e., RIB) based on the protocol preferences for each router.

2. Modeling packet forwarding on data plane. Second, the verifiers compute forwarding tables (i.e., FIB) for each router to determine the output port(s) for each incoming packet. To this end, they incorporate data-plane rules (e.g., access-control lists) in addition to the routing tables. The tables labeled Ⓢ and ① in Figure 2.4 show simplified examples of the resulting forwarding tables.

3. Verifying given specifications on the models: Finally, the verifiers checks whether each forwarding path requirement in given specifications are satisfied on the models. The verification algorithms depend on the underlying approach of the verifiers (e.g., based on graph theory or SMT-solver). We describe the algorithm details in §2.3.2.

Benefit. Control plane verification enables network operators to check whether router configurations satisfy their policies before deploying them to the network. This is important because router misconfigurations are a leading cause of network failures. Accordingly, control plane verification is a promising way to prevent configuration-induced failures and to help identify the specific configuration elements responsible for policy violations.

2.3.2 Existing approaches

There are four types of control plane verification: Simulation-based verification, Formula-based verification, and Simulation and Formula-based verification.

Simulation-based verification. Simulation-based control plane verification models the route computation of control plane protocols as even-driven [32,35–38,79–81]. For each advertised prefix, this approach continues the following processes until all the routes converge:

1. receives a route from neighboring nodes.
2. applies the route to import filters (e.g., route map), rank it, and apply it to outport filters.

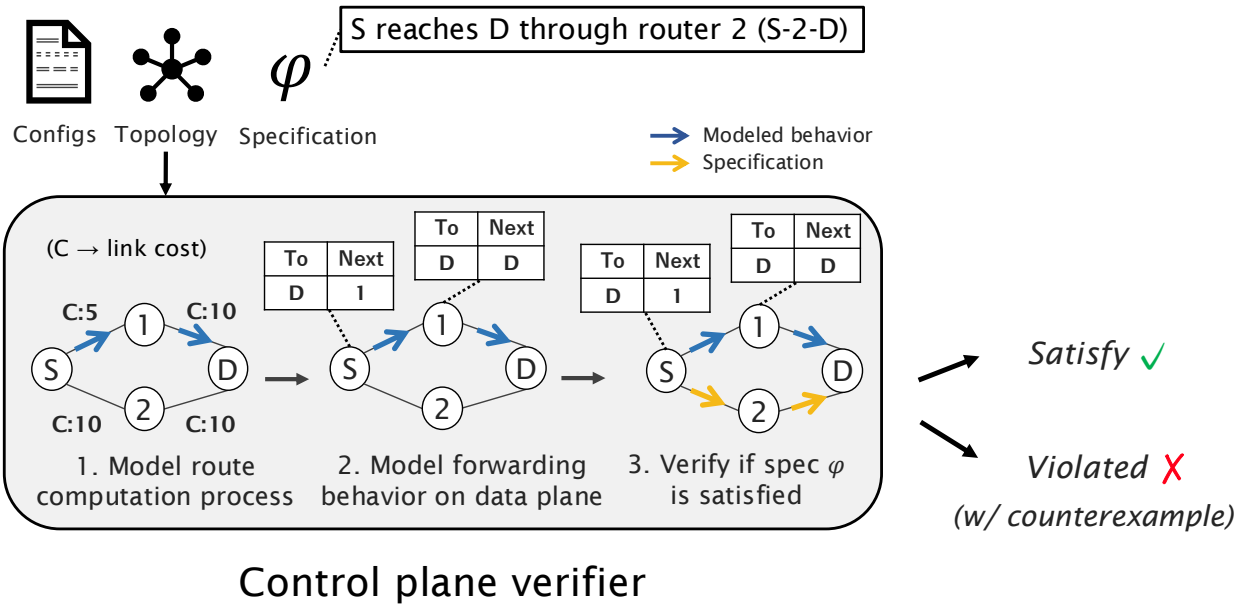


Figure 2.4: Control plane verification workflow.

3. advertises the best route to neighboring nodes if needed.

After the route convergence, the approach (1) creates routing tables for each router and forwarding tables based on the routing tables, and (2) models all the packet forwarding behaviors on a graph model as illustrated in Figure 2.4[1]. Then it uses general graph algorithms to verify whether the given specifications are satisfied on the model.

A key advantage of this approach is its flexibility in capturing real-world protocol semantics. For example, production route-map configurations often include regular expressions. Event-driven simulation can incorporate such constructs directly, whereas formula-based approaches may struggle to model these semantics precisely [33]. Moreover, explicit simulation allows operators to inspect the computed routing and forwarding tables [32,33], which formula-based and purely graph-based verifiers typically do not expose.

Formula-based verification. Formula-based verification encodes control-plane route computation as satisfiability modulo theories (SMT) constraints and checks whether the given specifications hold using an SMT solver (e.g., Z3 [82]) [34,75–78]. Griffin et al. [83] theoretically showed that BGP route computation can be formalized as the Stable Paths Problem, and these paths can be represented by edge constraints on a graph. Building on this insight, several systems [34,75–78] translate these edge constraints into logical formulas, rather than explicitly simulating route exchanges. A key advantage of this approach is that SMT solvers can symbolically reason about multiple routing messages and network conditions (e.g., arbitrary link failures) within a single model. For instance, Minesweeper [34] automatically finds a specific route announcement that triggers an IP prefix hijack.

Simulation and Formula-based verification. Simulation- and formula-based verification is a hybrid approach that combines event-driven simulation with SMT-based reasoning [9,25]. While formula-based verification can reason about all possible route announcements

[1]Tiramisu [37] and ARC [35] do not create forwarding tables. They only model route computation processes as a directed graph and verify given specifications on the graph model.

and link conditions, it often scales poorly [38]. To improve scalability while retaining this expressiveness, the hybrid approach incorporates symbolic link states (i.e., links being either up or down) into simulation-based verification. Hoyan [9,25], a control plane verifier used in Alibaba's global WAN, adopts this hybrid approach to identify unexpected reachability violations under arbitrary link failures.

2.4 Data plane verification

2.4.1 Overview

Data plane verification aims to verify whether packet forwarding behaviors on data plane satisfy given specifications [26,41–47,49,58,84–91]. We illustrate the overall verification flow in Figure 2.5. Unlike control plane verification, this approach directly reads packet forwarding rules, such as FIBs and ACLs, to model packet forwarding behaviors with a network topology. Since these rules are computed by (black-box) routing software in network device vendors, data plane verification can account for vendor-dependent behaviors (VDBs) that are difficult for control plane verification to capture [9,25]. For verification, the approach uses general graph algorithms to check whether each packet forwarding behavior on the model satisfies given specifications.

2.4.2 Existing approaches

Static verification. Early work on data plane verification focuses on static analysis of packet-forwarding rules [49,58,84–88]. Given a snapshot of forwarding state, these systems model forwarding behavior using logical formulas or Datalog [49,85], and then check whether the modeled behavior satisfies the specifications using techniques such as BDDs [92] and SAT/SMT solvers [82]. For example, SymNet and VMN statically verify forwarding behavior in the presence of stateful network functions such as NAT [87,88]. While static verification can handle both stateless and stateful behaviors, it is not designed for real-time (e.g., sub-millisecond) verification of forwarding changes.

Realtime verification. To support real-time verification, several systems incrementally check specifications in response to data-plane changes, such as FIB updates and link failures [41–45,47,50,52,90,91,93]. A key idea is to precompute packet forwarding equivalence classes (FECs) to reduce the number of packet headers that must be examined in each verification cycle. Intuitively, an FEC is a set of packets that follow the same forwarding behavior (e.g., the same forwarding path) in the network. For instance, Veriflow, Delta-net, and Libra create the prefix range-based FECs [26,45,47]. To further minimize the number of FECs, AP [41] and APKeep [43] define FECs based on port-level forwarding behaviors using atomic predicates. To handle header transformation in real-time verification, APT [42] and Katra [44] incorporate such transformations into the definition of FECs. For instance, Katra models header transformations as stack operations on a push-down automaton-based data plane model. This FEC-based approach achieves the real-time constraints required for operational networks, significantly outperforming offline approaches [48]. On the other hand, supporting stateful packet processing remains an open problem for this approach.

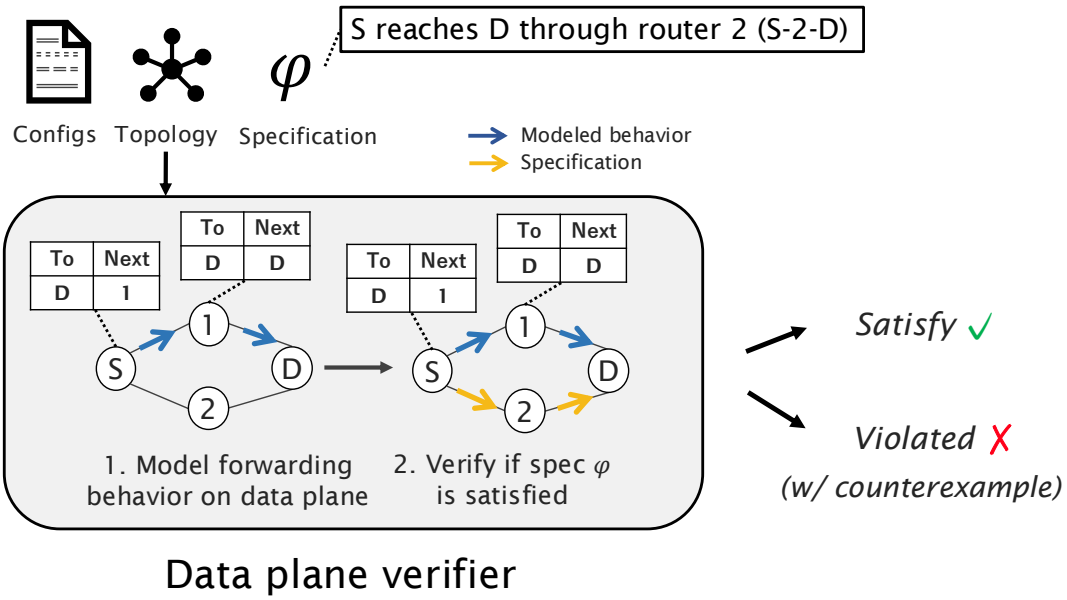


Figure 2.5: Data plane verification workflow.

Distributed verification. While the above real-time DPV tools typically rely on a centralized verifier, this architecture can become a scalability bottleneck and a single point of failure in large networks. Tulkun addresses this by offloading DPV to network devices: it transforms invariant checking into a counting problem on a DAG (DPVNet) and decomposes verification into lightweight on-device counting tasks, with devices exchanging results via a distributed verification messaging protocol [93]. This distributed, on-device design reduces dependence on a reliable management network and enables scalable data-plane checking across WAN/LAN/DC deployments.

2.5 Configuration repair

2.5.1 Overview

Configuration repair aims to safely automate router configuration changes in day-to-day network operations, helping prevent network failures caused by misconfigurations [29,94–106]. Figure 2.6 illustrates the overall repair workflow. It takes three inputs: router configurations, network topology, and specifications. The specifications for repair describe how network operators intend to change existing routing/forwarding policies or implement new ones. They are usually expressed as a set of routing paths using a regular expression with path preference.

To compute configuration changes that satisfy the specifications, repair tools typically proceed in three steps. First, they model route computation for the routing protocols configured in the network (similar to control plane verification). Second, they search a large configuration parameter space to infer new parameter values that satisfy the specifications. We explain the algorithm details to infer the parameters in §2.5.2. Finally, the repair tools output a set of router configurations that satisfy the given specifications.

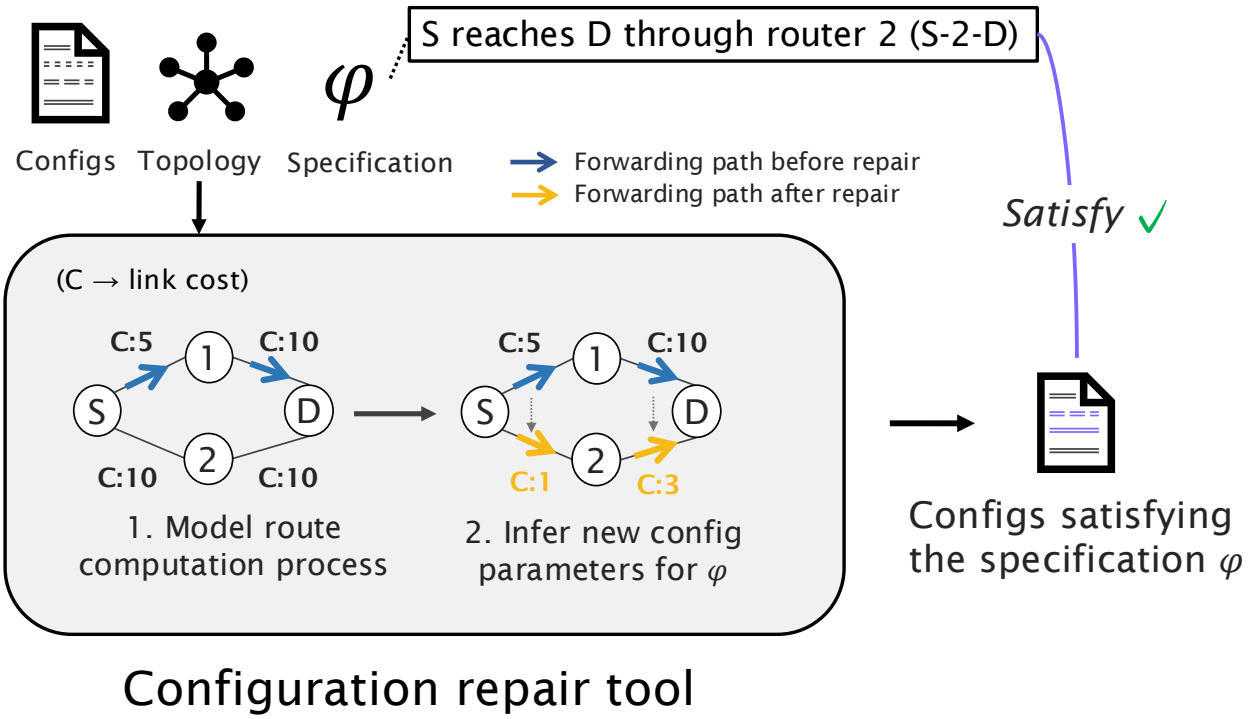


Figure 2.6: The overview of the configuration repair workflow.

2.5.2 Existing approaches

There are four types of underlying algorithms to infer new configuration parameters for repair (i.e., second step in Figure 2.6): formula-based repair, control-plane simulation-based repair, software testing method-inspired repair, and hybrid using a large language model (LLM) and formula-based repair.

Formula-based repair: This approach computes the new configuration parameters using an SMT-solver by encoding the route computation processes for each protocol and specifications into logical formulas (as with formula-based control plane verification) [97–99]. In this approach, network operators make a part of the parameters *holes* or *symbolic values*. Then an SMT-solver finds concrete parameters for each hole or symbolic value on the basis of the encoded route computation models.

Simulation-based repair: This approach (1) considers the given specifications as invariants in simulating the route computation processes and (2) computes why they are not preserved in given configurations [29]. As with simulation-based control plane verification, the approach runs the route computation simulation with the configurations. During the simulations, it checks whether routing paths in the specifications are not violated. If the violations happen, the approach computes new configuration parameters that do not introduce the violations.

Software testing method-inspired repair: This approach leverages the localize-fix-validate approach, which is inspired by software testing methods [96]. It takes three steps (localize, fix, and validate) for the repair. The first step calculates the suspiciousness score for all configuration lines to localize the suspicious ones using Spectrum-Based Fault Localization (SBFL) [107,108]. Second, the approach applies configuration changes to the

Table 2.1: Comparison of network testing, verification, and repair approaches

Approach	Input	Focus	Key Benefit
Network testing and monitoring	Synthetic probes or user traffic	End-to-end performance and reachability	Detect hardware & gray failures
Control plane verification	Router configurations and topology	Network properties computed from configs	Verify misconfis. before deployment
Data plane verification	Forwarding rules and topology	Network properties computed from rules	Verify forwarding states in realtime
Configuration repair	Configurations and change intents	Automated generation of configuration patches	Safely automates config changes

suspicious lines using the pre-defined change operators derived from real-world incident results. Third, it validates whether the changes satisfy given specifications using the off-the-shelf network verifier [36]. These three steps continue until all the specification violations are fixed. Note that this approach requires a real-world history of configuration changes to generate configuration change plans, unlike the other approaches.

LLM and Formula hybrid repair: This approach leverages the inference capability of LLM to generate new configuration parameters and the correctness of network verifiers to validate the new parameters [94,109]. While LLM can efficiently infer plausible configurations to satisfy given specifications, the outputs may include incorrect results, which could lead to misconfigurations and network failures. Therefore, this approach checks whether the resulting configurations satisfy the specifications using the network verifiers.

2.6 Summary

This chapter has provided a comprehensive overview of fundamental network concepts and the approaches used to ensure network reliability. The following summary (also in Table 2.1) highlights the roles and characteristics of each approach:

- Network Testing and Monitoring (§2.2): This approach treats the network as a black box. It relies on active probing (e.g., ping, traceroute) or passive monitoring of real user traffic to measure end-to-end metrics such as latency, availability, and packet loss. Unlike verification methods, it does not require an explicit network model and is primarily used to track service-level agreements (SLAs) in heterogeneous environments.
- Control Plane Verification (§2.3): This approach focuses on router configurations and topology to model route computation processes and verify its convergent states. Its primary advantage is to identify potential failures in router configurations before deployment, preventing configuration-induced failures from ever reaching the production network.

- Data Plane Verification (§2.4): Unlike control plane analysis, this approach directly inspects active forwarding rules (e.g., FIBs and ACLs). This allows it to verify the actual state of the network and account for vendor-dependent behaviors (VDBs) that might be missed by control plane models. It is often used for real-time checking of forwarding changes and link failures.
- Configuration Repair (§2.5): While the verification methods focus on identifying policy violations, configuration repair aims to safely automate configuration changes. It automatically generates patches that satisfy given high-level network change intents of operators by searching through configuration parameter spaces.

Chapter 3

Specification Refinement for Automatic Configuration Repair

In this chapter, we introduce a specification refinement approach that automatically creates correct specifications for repair only with the initial operator's intents.

3.1 Introduction

Configuration changes on network devices are a daily task for network operators. Operators perform tens to hundreds of configuration changes weekly or monthly to implement new routing and forwarding policies or modify existing ones [27,36,110–112]. However, misconfigurations can cause severe network failures, sometimes resulting in significant downtime [8,9,22,96,113–115].

Configuration repair promises to safely automate configuration changes to prevent the misconfigurations [29,94,96–103,116]. Given high-level network change intents (usually described by forwarding path requirements), the repair tools automatically generate patches that indicate how to modify current configurations to satisfy the intents. Recent techniques leverage heuristic search strategies [29,98,99,101] or combine formal methods and large language models (LLMs) [94,100,103] for repair to achieve scalability and reliability in large and complex networks.

Despite these advancements, there is a major barrier to real-world deployment. Repair tools could cause unexpected forwarding path changes not described by the change intents. We refer to such unexpected changes as side effects in repair (the definition in §3.2.2). Figure 3.1 illustrates a simple example of the side effect. In this network, which uses shortest path routing (e.g., OSPF), the repair specification describing a change intent dictates a forwarding path change from router S to D. The new path must traverse router B instead of router A. Given the inputs, a repair tool generates a patch to reduce two link costs: (S, B) and (B, D). The cost change satisfies the change intent; however, it also unintentionally alters the forwarding path from router A to D. This example may seem trivial (albeit a real one as shown in §3.6), but the risk of such side effects impedes the widespread adoption of repair tools because forwarding paths can be changed without the operator's awareness.

To prevent side effects, network operators must constrain repair tools via specifications

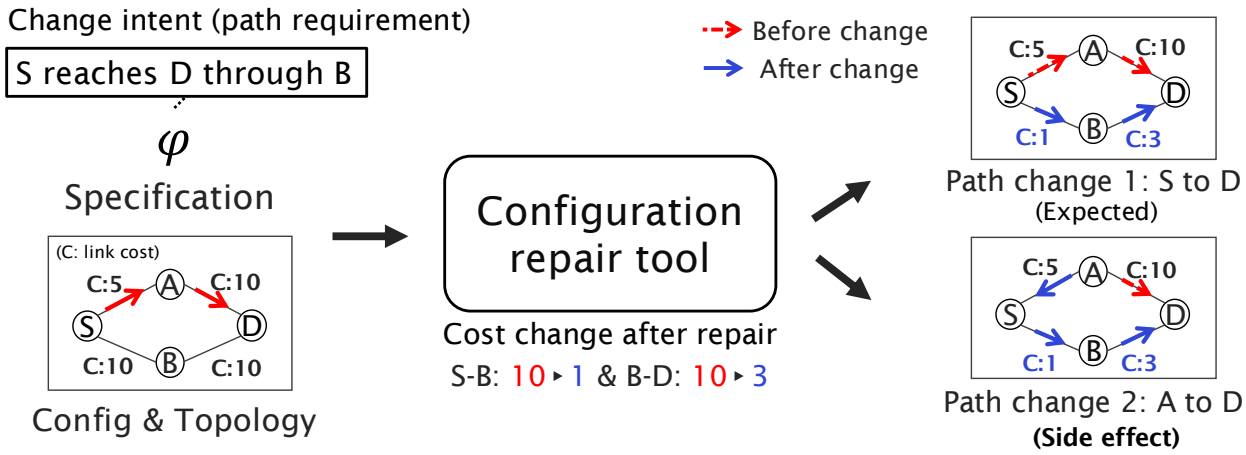


Figure 3.1: Example of side effect in configuration repair.

that completely preserve existing forwarding paths that can be affected by change intents. The repair tools use the specifications as search constraints in configuration parameter spaces to infer new parameters satisfying the change intents. Therefore, these tools do not inherently guarantee that existing forwarding paths not described in the specifications remain unchanged.

However, creating such complete specifications imposes a substantial burden on operators [33,55,56]. On the scale and complexity of today's networks, it is not straightforward to predict which existing forwarding paths can be impacted by new change intents. In addition, translating the high-level intents into specification formats required for repair tools may not always yield a one-to-one mapping. Specification mining helps extract all existing forwarding paths from configurations as specifications [56,57,117–119]. However, side effects can still be introduced when conflicts between new change intents and existing forwarding requirements occur (details are discussed in §3.2.3).

We propose ISR, an augmentative framework for repair tools that shifts the burden of preventing side effects from network operators to an iterative specification refinement. ISR externally guides these tools to prevent side effects in configurations after repair by refining specifications. The workflow (Figure 3.2) runs iteratively (§3.4): ISR identifies side effects in repaired configurations (Side effect diagnosis in §3.4.1 and §3.4.2), refines existing specifications to preserve original forwarding paths prior to these side effects (Specification refinement in §3.4.3), and re-executes the repair with the refined specifications (§3.4.4). This iterative process incrementally constrains the repair tools to find configuration parameters that satisfy change intents without undesired side effects.

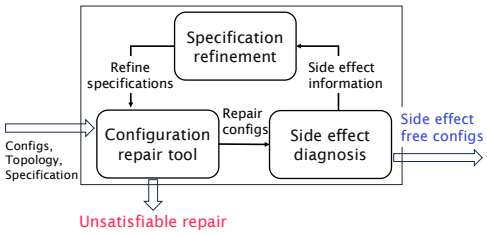


Figure 3.2: High-level diagram of ISR.

Consequently, ISR reduces the operator's burden to two essential tasks in repair: (1) defining new network change intents, where operators can focus on what they want to change while leaving the side effect prevention to ISR; and (2) determining the priority between the change intents and unavoidable side effects, where, if the intent cannot be satisfied without

introducing a specific side effect (unsatisfiable repair in Figure 3.2), operators decide whether to revise the intent or relax existing forwarding path requirements to accept the side effect. ISR transforms the workflow from manual side effect prevention to high-level decision-making to achieve the intents.

We fully implement ISR for OSPF and BGP configuration repair. ISR treats repair tools as black boxes, enabling it to work with a wide range of repair tools regardless of their underlying mechanisms (e.g., SMT or LLM). Given repair specifications, network topology, and the configurations before and after repair, ISR first identifies side effects by tracing the differences in route computation before and after repair using control plane simulation [32,33,36,37,78]. It then generates new specifications that preserve original forwarding paths prior to these side effects (e.g., router A directly reaches D in Figure 3.1). To design ISR, we make the three technical contributions:

1. Causality inference of side effect (§3.4.2). If repair becomes unsatisfiable due to specification conflicts, operators must decide whether to revise new change intents or relax existing forwarding path requirements to accept unavoidable side effects for the intents. Making this decision requires understanding the underlying causality: (1) which new change intents induce which configuration changes, and (2) which configuration changes induce which side effects (because existing repair tools do not explain it [120]). To help this process, we develop a method to infer the causality by analyzing the differences in route computation before and after repair.

2. Specification minimization in refinement (§3.4.3). If a repair affects many forwarding paths, new specifications to prevent side effects can be huge, making specification conflicts more likely. To prevent unnecessary conflicts, we develop an algorithm to minimize the number of specifications added to the next repair. Our insight is that in common routing protocols (e.g., OSPF/BGP), any sub-path of a best path must also be optimal. For instance, if the best path from router S to D is $S \rightarrow A \rightarrow D$, the path from router A to D must be $A \rightarrow D$. Leveraging this property, we compute a minimal set of specifications by identifying the longest paths that subsume all required sub-paths.

3. Termination of the iterative refinement (§3.5). With repair tools and ISR, we formally prove that the iterative refinement process always converges to one of two final states: (1) generating configurations without introducing undesired side effects, or (2) the repair becomes unsatisfiable due to specification conflict. This proof is tool-agnostic under a reasonable assumption, ensuring the feasibility of ISR designed to support diverse repair tools.

We evaluate the effectiveness and scalability of ISR on real-world network topologies. First, we demonstrate that ISR successfully guides three different repair tools (SMT [99], Simulation/SMT [29], and LLM [94,116]) to generate configurations without undesired side effects in OSPF and BGP networks (§3.6). Second, we show that ISR is scalable, completing a refinement iteration within seconds for various numbers of side effects on the largest topologies with up to 200 routers (§3.7).

3.2 Background

3.2.1 Overview of Configuration Repair

Configuration repair aims to automatically generate configuration patches to satisfy high-level network change intents of operators (as illustrated in Figure 3.1) [29,94,96–103,116]. Given the configurations, network topology, and specifications describing the intents, repair tools (1) model route computation processes and (2) search for new parameters in the configurations to satisfy the specifications. Currently, there are several approaches to perform the parameter search, such as using SMT-solver [94,97–99], control plane simulation [29], and LLM [94,100,116].

Repair specifications to describe network change intents are expressed as a set of prioritized forwarding path requirements capturing practical network properties such as reachability and waypoint [1] [29,94,96–99,101,116]. For instance, a specification in Figure 3.1 requires a repair tool to forward packets from router S to D through router B. Repair tools use the specifications as search constraints in a huge parameter space of router configurations. If parameters that satisfy the constraints are found, the tools generate configuration patches with those parameters. Otherwise, they report that the specifications are unsatisfiable (i.e., no valid configuration exists).

3.2.2 Side Effect in Configuration Repair

While the risk of unintended changes in repair has been discussed in recent works [103,121], its definition varies across different contexts. In this paper, we define side effects in repair as best path changes of prefixes not covered in the specifications. Formally, let $P(C)$ be the set of all forwarding paths computed from a configuration set C. We denote the configuration sets before and after repair as C_{before} and C_{after}, respectively. Let S be the set of forwarding path requirements (denoted as p_s) as specifications for network change intents. The set of side effects (SE) is defined as follows:

$$SE = \{p_{new} \mid p_{new} \in (P(C_{after}) \setminus P(C_{before})), \quad \nexists p_s \in S \text{ s.t. } p_{new} \text{ is a sub-path of } p_s\}$$

For example, in Figure 3.1, the forwarding path change from router A to D is classified as a side effect because this change was not described in the specification.

Repair tools could introduce side effects, as their configuration parameter search in repair does not inherently guarantee that unspecified behaviors remain unchanged. ACR [96], a repair tool adopting a localize-fix-validate approach, attempts to prevent unexpected changes through verifying configurations after repair [36]. Despite its benefit, this verification also relies on specifications to determine whether each change is expected; consequently, it may overlook side effects not described in the specifications. The presence of side effects hinders the widespread adoption of the repair tools because the unspecified forwarding paths can change without the operator's awareness.

[1] While some repair tools have domain-specific languages for specifications [97,98,101,104,106], all the languages include routing/forwarding path requirements for synthesis and repair. We discuss different kinds of specifications in **??**.

3.2.3 Reducing the Burden of Specification Writing

To prevent side effects, network operators must create repair specifications that preserve all existing forwarding paths potentially impacted by new network change intents. However, creating such complete specifications is tremendously difficult for operators due to the scale and complexity of today's networks [55,56]. We introduce the two approaches that reduce the difficulty of creating the specifications and discuss their limitations to prevent side effects in repair.

Specification mining. Specification mining techniques extract all network properties (e.g., reachability and isolation) from given configurations and network topology in the form of specifications [56,57,115,117–119]. They are useful to create a comprehensive specification set that preserves all existing forwarding paths. Despite the benefit, side effects can still be introduced when a network change intent conflicts with the specifications. To resolve the conflict, operators must manually revise the change intent or relax forwarding requirements in the specifications to accept the side effects. However, the relaxed specification may fail to fully preserve other existing paths, allowing repair tools to introduce new side effects. Thus, even with specifications for all existing paths, the threat of side effects is not eliminated but shifted to the manual, error-prone conflict-resolution.

Relational specification. Rela [55] introduces *relational* specifications that describe the difference and similarity before and after configuration changes. For network verification, Rela accepts (1) "change" specifications that describe how traffic should be changed and (2) "no change" specifications that ensure the rest of the traffic remains unchanged. While useful for *verifying* these similarities and differences, it is difficult to apply relational specifications to the configuration repair process. To interpret relational specifications, the approach requires forwarding path sets from both before and after the changes. However, since repair is a process of modifying configurations to satisfy new network change intents, repair tools do not yet have access to the forwarding paths after the changes in repair, which cannot interpret the relational specifications.

3.3 Overview

The goal of this paper is to achieve configuration repair without undesired side effects. In this section, we introduce our key idea (§3.3.1) and the overall workflow (§3.3.2) to achieve the goal.

3.3.1 Key Idea

To achieve configuration repair that satisfies new network change intents without introducing undesired side effects, our approach is to *iteratively refine* given specifications for the change intents. Concretely, we (1) identify side effects in configurations after repair, (2) refine existing specifications by creating new path requirements to prevent the side effects, and (3) re-invoke a repair tool with the refined specifications. We repeat the three steps until no undesired side effects are found or the repair becomes unsatisfiable. The iterative steps create localized specifications to preserve original forwarding paths prior to the side effects. By imposing these specifications as new constraints on the configuration parameter search in repair, this

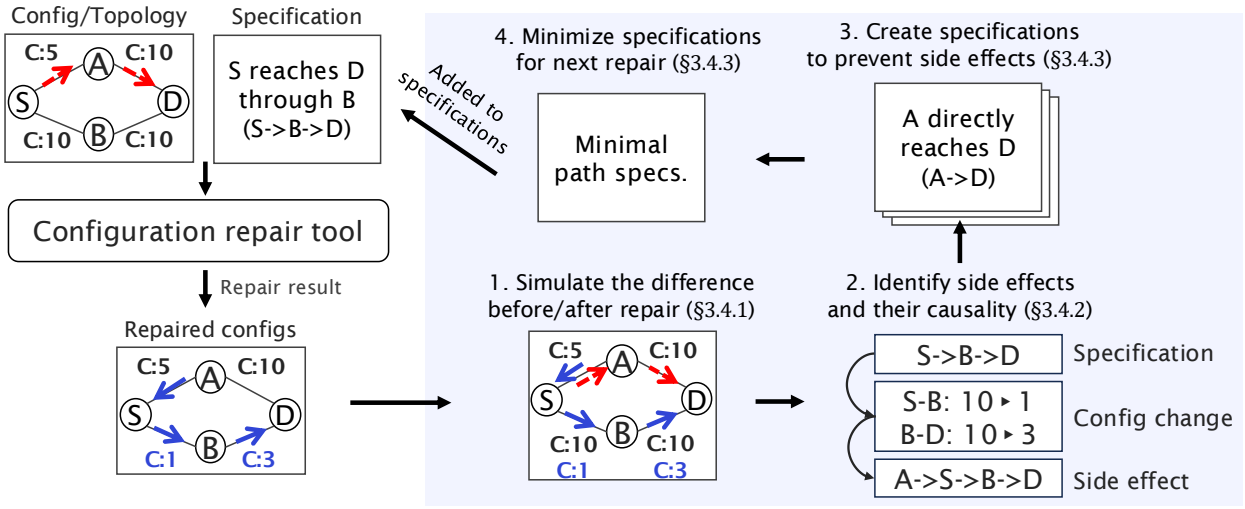


Figure 3.3: Overall workflow of iterative specification refinement with side effect diagnosis in ISR. For simplicity, we only illustrate forwarding paths from router S and router A to router D.

approach guides repair tools to generate configurations that satisfy the change intents without introducing undesired side effects.

This approach enables operators to focus on two essential tasks in repair: (1) defining the change intents and (2) determining the priority between the change intents and unavoidable side effects. First, operators need only specify the change intents as initial repair specifications, as this approach guides repair tools to prevent side effects by refining the repair specifications. Second, when the change intents cannot be satisfied without introducing a specific side effect, operators decide whether to revise the intents or relax existing forwarding path requirements to accept the side effect. Crucially, this approach prevents the manual specification changes from introducing further side effects, unlike leveraging the existing approaches (§3.2.3).

3.3.2 Workflow with Example

We propose ISR, a framework that performs the iterative specification refinement for repair tools. To work in conjunction with various types of repair tools, ISR takes general input and output of repair as its input: (1-2) router configurations before and after repair, (3) network topology, and (4) specifications for repair. As with these tools, ISR accepts the specifications expressed as a set of forwarding paths using a regular expression with path preference. For instance, a requirement "router S should reach D through router B" is expressed as $<P, S \rightarrow B \rightarrow D>$ where P is an IP prefix.

Figure 3.3 illustrates the iterative refinement for the example in Figure 3.1. The refinement is initiated when repair tools generate configurations that satisfy the specifications. This process consists of three steps: (1) side effect diagnosis, (2) specification refinement, and (3) re-invoking a repair tool.

1. Side effect diagnosis (§3.4.1 & §3.4.2). In the first step, ISR identifies side effects in configurations after repair (processes 1 and 2 in Figure 3.3). This step consists of two phases: (1) simulating the differences in the route computation before and after

repair using control plane simulation [9,32,33,35–37,78] and (2) identifying side effects and inferring their causality based on the differences. This causality inference is crucial in practice because existing repair tools do not explain which configuration changes are introduced to satisfy which specification [120]. Concretely, ISR infers the following two relations: (1) which specifications induce which configuration changes, and (2) which configuration changes introduce which side effects. This inference helps operators understand why side effects happen and reconsider repair specifications when resolving specification conflicts.

In the example of Figure 3.3, ISR first simulates the shortest path calculation in OSPF semantics. It then detects the two forwarding path changes: paths from router S to D and from router A to D. Since the latter path change is unspecified, ISR evaluates it as a side effect. It then infers (1) which cost changes induce the side effect and (2) which specification introduces the cost changes.

Note that while ISR defaults to preventing all detected side effects, operators may accept minor side effects for change intents during the refinement. ISR also allows operators to assess whether each side effect is acceptable. We explain this human-in-the-loop refinement in §A.1.

2. Specification refinement (§3.4.3). In the second step, ISR refines repair specifications to prevent side effects identified in the first step (processes 3 and 4 in Figure 3.3). To this end, it creates new path requirements that preserve original forwarding paths prior to the side effects. However, a naive translation of these requirements into repair specifications can lead to a huge number of specifications, making conflicts more likely. Therefore, leveraging the insight that any sub-path of an optimal path must also be optimal in common network protocols, ISR computes the new, minimal requirements by identifying the longest paths that subsume all required sub-paths. This minimal set is then integrated into the specifications for the next repair.

In our example, ISR creates a new path requirement that preserves the direct path from router A to D (i.e., $A \rightarrow D$). Since the number of specifications is minimal in this case, ISR adds the requirement to the specifications for the next repair.

3. Re-invoking repair tool (§3.4.4). Finally, ISR re-invokes the repair tool using the refined specifications. These specifications guide the repair tool to find configuration parameters to satisfy the new network change intents while preventing the side effects identified in the previous repair. For instance, the refined specifications in Figure 3.3 constrain the repair tool to search for link cost updates that satisfy the initial intent $S \rightarrow B \rightarrow D$ while keeping the path $A \rightarrow D$ unchanged.

If the repair tool returns a configuration set, ISR proceeds to side effect diagnosis. This process terminates if no side effects are found; otherwise, it repeats the refinement steps. Conversely, if the repair tool returns an unsat result, the process terminates and reports a specification conflict. To resolve it, operators must decide whether to revise the change intents or relax the forwarding path requirements that prevent side effects in the specifications. When revising the change intents, ISR restarts the repair from scratch with the intents. This is because the previously generated path requirements to prevent side effects are specific to the initial intents and may be irrelevant to the revised intents. If relaxing the path requirements, ISR incorporates them into the specifications as additional change intents and re-invokes the repair tool to find a feasible configuration.

3.4 Iterative Specification Refinement with Side Effect Diagnosis

In this section, we explain the details of side effect diagnosis (§3.4.1 and §3.4.2), specification refinement (§3.4.3), and the next repair iteration (§3.4.4).

3.4.1 Simulating the Differences in Route Computation before and after Repair

To identify side effects, ISR first computes the route computation differences before and after repair (process 1 in Figure 3.3). To achieve this, ISR simulates the route computation and its resulting forwarding paths with configurations before and after repair by leveraging control plane simulation [9,32,33,35–37,121]. In the simulation, we generate a single trace, referred to as a differential trace [121], that captures where and why each best path differs. It records (1) all forwarding paths before and after repair whose attributes or metrics have changed, and (2) the configuration changes introducing these changes. We leverage it to identify side effects and their causality (§3.4.2).

ISR constructs the differential trace as a directed acyclic graph where each node contains RIBs before and after repair. Each edge has two colors to represent the route propagation before and after repair. Based on this graph, ISR simulates route computation in an event-driven manner [33,122].

Figure 3.4 illustrates the differential trace of the repair scenario in Figure 3.3. ISR runs the route computation simulation for each advertised prefix using the configurations before and after repair, respectively. While these two processes are separately executed, their simulation traces are recorded on the same differential trace using two edge colors (red and blue edge colors in Figure 3.4). When ISR encounters the configuration differences (e.g., the change of link cost between router S and B) in the simulation, it records the differences on the corresponding edges on the trace. For instance, the resulting trace for this repair scenario contains (1) two forwarding path changes from router S to D and from router A to D, and (2) the link cost changes that induced these path changes (specifically, the cost changes for links S-B and B-D).

3.4.2 Side Effect Diagnosis

Second, ISR identifies side effects and infers their causality by searching the differential trace.

1. Identifying side effects. This process consists of two main steps. First, ISR compares the best path in each node's RIB before and after the repair on the differential trace (e.g., OSPF RIBs in Figure 3.4). Second, whenever a best path change is detected, ISR checks whether the new path is explicitly covered by the repair specifications. Any path change not in the specifications is classified as a side effect. In our example (Figure 3.4), while the path change from router S to D is defined in the specification, the change from router A to D is unspecified, and is thus classified as a side effect.

2. Tracking which specification induces which configuration changes. Next, ISR infers which specifications induce the configuration changes generated by repair tools.

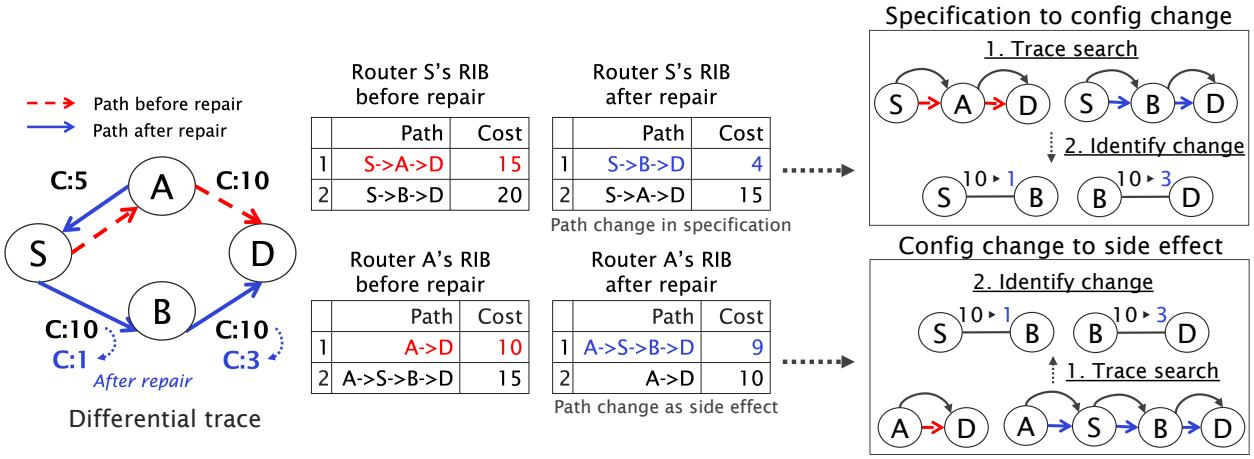


Figure 3.4: Tracking the causality of side effects using differential trace for Figure 3.3.

This relation is not straightforward for ISR (also for network operators) because existing repair tools do not explain which configuration changes were introduced to satisfy which specification [120]. To infer this, ISR leverages the strategies repair tools employ to satisfy a new path requirement. The tools (1) directly promote the path in specifications to make it the best path after repair (e.g., by lowering its cost or increasing its local-preference), or (2) indirectly achieve this by deeming the other paths (e.g., by increasing its cost), thereby causing the specified path to be selected as best, or (3) use a combination of both strategies. Therefore, these configuration changes must necessarily be applied to routers that are on the best path before repair, the other paths after repair, or both.

Based on this insight, ISR first identifies why the forwarding paths in the specifications become the best paths after repair by comparing each node's RIB before and after repair. Since the RIB stores all candidate paths with the metrics and attributes used in best-path selection (e.g., OSPF link cost and BGP local preference), this comparison reveals every path and its metrics/attributes involved in the best path change—including not only the best paths before and after repair but also other lower-ranked paths considered in the selection process. Finally, ISR searches these paths on the trace to pinpoint the specific configuration changes responsible for these metrics and attributes.

In our example (Figure 3.4), ISR first compares the RIBs of router S before and after repair, since the repair specification dictates the path change from router S to D. This comparison reveals that the path change was induced by the link cost change(s). Consequently, ISR searches the differential trace along the two relevant paths in the RIBs: (1) $S \to A \to D$ and (2) $S \to B \to D$. It identifies the two link cost changes that induced the path change from router S to D: the cost changes on links S-B and B-D. Finally, these changes are linked to the specification ($S \to B \to D$).

3. Tracking which configuration changes induce which side effects. ISR then infers which configuration changes induce each side effect by searching the differential trace. In contrast to the previous second process, this process identifies why the best paths before repair are no longer the best after repair. To this end, ISR first compares the RIBs of each node where the best path change was classified as a side effect, to understand the primary reasons for the change (e.g., a cost increase, a withdrawn route, or a local preference change).

It then identifies corresponding configuration changes by searching the differential trace, as in the previous process.

In our example (Figure 3.4), comparing the router A's RIBs reveals that the side effect was induced by link cost change(s). Accordingly, ISR searches the differential trace along the two paths in the RIBs ($A \to D$ and $A \to S \to B \to D$) and locates the link cost changes on links S-B and B-D. Since both cost changes contributed to the path change, ISR links them to the side effect.

4. Creating a map of the causality of side effects. After tracking the two relations, ISR concatenates them and creates maps to explain them (the map of process 2 in Figure 3.3). Configuration changes inducing a side effect may be introduced by multiple repair specifications. Therefore, ISR (1) identifies all the specifications that introduce configuration changes for each side effect and (2) creates a mapping from those specifications to the side effect. These (many-to-many) relationships are managed efficiently using a hash map.

Multiple protocol dependencies. In multiple protocol networks, routing decisions are hierarchical; for instance, BGP route selection often depends on OSPF for resolving next-hop reachability. Our simulation respects this dependency by resolving OSPF states before computing BGP routes (similar to [36]). Accordingly, we adopt a dependency-aware search for causality inference: we group route changes by protocol and schedule the analysis of OSPF changes before BGP changes. This sequential processing ensures that ISR correctly identifies cases where a BGP best path change is caused by an underlying OSPF configuration change, allowing for accurate mapping.

3.4.3 Specification Refinement

After the side effect diagnosis, ISR refines repair specifications by creating new path requirements to prevent the side effects for the next repair. To this end, ISR creates the path requirements specifying original forwarding paths prior to the side effects (e.g., $A \to D$ in Figure 3.3). These requirements are then incorporated into the specification set, acting as new constraints that guide the configuration parameter searches in repair to preserve the original forwarding paths.

However, naively adding such path requirements can result in a prohibitively large specification, especially when many side effects are generated in repair. Such large specifications are prone to cause unnecessary conflicts. To prevent them, we *minimize* the number of new path requirements for the next repair. Our insight for the minimization is that any sub-path of a best path must also be optimal in common network protocols (e.g., BGP and OSPF). For instance, when the best path from router A to D is $A \to S \to B \to D$, the routers in the path (i.e., router S and B) also select the subpaths as their best paths to reach router D (e.g., $S \to B \to D$). The single path requirement, $A \to S \to B \to D$, covers the corresponding subpath requirements (e.g., the path from S to D must be $S \to B \to D$) and guides repair tools to satisfy both requirements. Therefore, finding such path requirements that include the other subpaths can reduce the total number of specifications while enforcing the same path requirements to the repair tools. Note that in BGP, the minimization is executed independently for each advertised prefix because of path attributes/policies per prefix.

Based on the insight, we formulate the minimization of forwarding path requirements as a minimal containment path cover problem.

Definition 1 (Containment Relation)**.** For two directed paths p, q in a path set $\mathcal{P}$, we write $p \sqsubseteq q$ if p is a contiguous subpath of q. That is, the sequence of edges of p appears consecutively in q.

Given a finite path set $\mathcal{P}$ on network G, we find the smallest subset $\mathcal{S} \subseteq \mathcal{P}$ such that for every $p \in \mathcal{P}$, there exists $s \in \mathcal{S}$ with $p \sqsubseteq s$. We call such an $\mathcal{S}$ a *minimal containment path cover* of $\mathcal{P}$.

Theorem 1 (Characterization of the Minimal Cover)**.** *Let* $M = \{p \in \mathcal{P} \mid \nexists q \in \mathcal{P}, q \neq p, p \sqsubseteq q\}$. *Then* M *is the unique minimal containment path cover of* $\mathcal{P}$.

Proof. (*Sufficiency*) Every path $p \in \mathcal{P}$ is either in M or contained in some $q \in M$, since all strictly contained paths are removed by construction.
(*Necessity*) Suppose $\mathcal{S}$ is any cover omitting a path $m \in M$. Then m is not contained in any other path of $\mathcal{P}$, contradicting the definition of a cover. Hence, $M \subseteq \mathcal{S}$, and M is the smallest cover. □

Algorithm 1: Longest-Path-First Algorithm (Computing the Minimal Cover M)

Input : $\mathcal{P}_{\text{set}}$, a set of paths $\{P_1, \ldots, P_n\}$
Output : M, the minimal containment path cover

```
1  Function ComputeMinimalCover(P_set)
2      P_sorted ← SortByLengthDescending(P_set)
3      M ← ∅
4      CoveredPaths ← ∅
5      for P_i ∈ P_sorted do
6          if P_i ∉ CoveredPaths then
7              M ← M ∪ {P_i}
8              for P_j ∈ P_set do
9                  if IsSubpath(P_j, P_i) then
10                     CoveredPaths ← CoveredPaths ∪ {P_j}
11                 end
12             end
13         end
14     end
15     return M

Function IsSubpath(P_A, P_B)
    if vertex sequence of P_A is a contiguous subsequence of P_B then
        return true
    end
    else
        return false
    end
```

We compute the unique minimal containment path cover, $\mathcal{M}$, using an efficient filtering procedure in Algorithm 1. The core strategy is to process paths in a specific order that allows it to use longer paths to "filter out" their shorter, contained subpaths. The algorithm begins by sorting all paths in the input set $\mathcal{P}_{set}$ in descending order of their length (Line 2). It proceeds by iterating through the sorted paths $\mathcal{P}_i$ (Line 5). For each path $\mathcal{P}_i$, it first checks if $\mathcal{P}_i$ has already been marked as covered (Line 6). If $\mathcal{P}_i$ is not in $CoveredPaths$, $\mathcal{P}_i$ is not contained by any path processed so far. Therefore, $\mathcal{P}_i$ must be a maximal path and is added to the solution set M (Line 7). Then this algorithm updates $CoveredPaths$ (Lines 8 to 10). Over the original path set $\mathcal{P}_{set}$, it finds all path $\mathcal{P}_j$ that are subpaths of $\mathcal{P}_i$ (Lines 8 to 9). All such paths are added to $CoveredPaths$ (Line 10).

The computational complexity of the algorithm is $\mathcal{O}(n^2 L_{\max})$, where $n = |\mathcal{P}|$ and $L_{\max}$ is the maximum path length. For large-scale networks (e.g., over a thousand network devices), we may reduce the complexity to nearly linear in the total input size by employing advanced data structures (e.g., a generalized suffix tree). We provide the completeness and soundness proof in §A.2.

3.4.4 To the Next Repair Iteration

After the specification refinement, the minimal path requirements are integrated with the specifications in the previous repair. This specification set contains both the initial network change intents and the newly generated constraints that preserve original forwarding paths prior to side effects.

ISR then re-invokes a repair tool to find configuration parameters that satisfy these specifications. The result of this repair dictates the next step in the workflow, leading to one of three outcomes:

1. Success: If the repair tool returns a valid configuration and the side effect diagnosis step finds no new side effects, ISR successfully completes the refinement process. The resulting configuration is confirmed to satisfy the network change intents without introducing undesired side effects.

2. Further refinement: If the repair tool returns a valid configuration but the diagnosis step identifies new side effects, the entire iterative refinement is repeated to prevent them.

3. Failure: If the repair tool returns unsatisfiable, the refinement terminates. This indicates a conflict between the change intents and the forwarding path requirements to prevent side effects. ISR reports the causality of this conflict (§3.4.2) to operators for the conflict resolution.

3.5 Termination of Iterative Specification Refinement

Given the iterative nature of the specification refinement, a natural question is whether the refinement is guaranteed to terminate. We formally prove that the refinement process always terminates, resulting in either the *Success* or *Failure* state described above.

3.5.1 Preliminary

We first define the core components of our iterative process and our assumption.

Forwarding path. We define the forwarding path set (F_{all}) as the set of all possible loop-free forwarding paths in a network, bounded by its nodes, links, and prefixes.

Specification. Our specification refinement operates over a specification set (S_i) at iteration i. S_i is composed of two disjoint parts. The first part is the set of specifications for change intents ($S_{\text{intent},i}$). Initially, $S_{\text{intent},0} = S_{\text{init}}$, where S_{init} is the set of specifications for the initial intents. If a side effect is accepted when resolving a specification conflict (§3.3.2), the corresponding path requirement (permitting the side effect path) is added to a set of accepted side effects ($S_{\text{accepted},i}$). Thus, the set of change intents at iteration i is updated as $S_{\text{intent},i} = S_{\text{init}} \cup S_{\text{accepted},i}$. The second part is the preserved set $S_{\text{preserve},i}$: Let $S_{\text{preserve},i}$ be the set of specifications added by the refinement to prevent side effects, accumulated up to iteration i. The total specification set is given by $S_i = S_{\text{intent},i} \cup S_{\text{preserve},i}$.

Assumption for repair tools. We assume that repair tools generate router configurations that satisfy given specifications. This assumption holds for repair tools based on formal methods [29,96–99,101,102]. However, LLM-based repair tools [94,100,116] sometimes generate configurations that do not satisfy some specifications [95,103] (as also shown in §3.6.1). Thus,

we must validate the configurations generated by these tools using network verification tools [33,52,78,123,124].

Formally, let R be a repair tool. We model R as a function that maps a specification set S (from the space of all possible specifications $\mathcal{S}$) to either a configuration C (from the space of all configurations $\mathcal{C}$) or an *Unsatisfiable* result:

$$R : \mathcal{S} \to \mathcal{C} \cup \{\text{Unsatisfiable}\}$$

The function R guarantees that for any $S_i \in \mathcal{S}$:

1. If $R(S_i) = C_i \in \mathcal{C}$, then $C_i \models S_i$ (the returned configuration satisfies the specifications).
2. If $R(S_i) = \text{Unsatisfiable}$, then $\nexists C \in \mathcal{C}$ such that $C \models S_i$ (no configuration exists that satisfies the specifications).

3.5.2 Termination Proof

Theorem 2 (Termination)**.** *The iterative specification refinement terminates in one of two states: (1) Success (a valid configuration is found) or (2) Failure (the repair becomes unsatisfiable).*

Proof. Let F_{all} be the finite universe of all forwarding paths in the network. We define $\mathcal{P}_i \subseteq F_{\text{all}}$ as the set of forwarding paths that are constrained by the specification set S_i at iteration i. The iterative refinement proceeds from iteration i to $i+1$ only if the diagnosis identifies a non-empty set of side effects $SE_i \neq \emptyset$. A side effect corresponds to a forwarding path $p \in F_{\text{all}}$ that deviates from the expected state. If $R(S_i) = \text{Unsatisfiable}$, the repair terminates immediately with **Failure**; thus, in the following, we assume $R(S_i) = C_i \in \mathcal{C}$. Based on the assumption that R produces valid configurations, the generated configuration $C_i = R(S_i)$ satisfies all specifications in S_i (i.e., $C_i \models S_i$). This implies that any path q that is already constrained by S_i (i.e., $q \in \mathcal{P}_i$) must satisfy its corresponding specification and cannot appear as a side effect. Therefore, any identified side effect $p \in SE_i$ must belong to the set of unconstrained paths, meaning $p \notin \mathcal{P}_i$. In each iteration, a new specification set S_{new} is generated to address the paths in SE_i. Regardless of whether the refinement chooses to *preserve* the original path (adding to S_{preserve}) or *accept* the side effect (adding to S_{accepted}), the new specification imposes a constraint on the path p. Consequently, the set of constrained paths expands strictly: $\mathcal{P}_{i+1} = \mathcal{P}_i \cup \{p \mid p \in SE_i\}$. Since $\mathcal{P}_i$ is a subset of the finite set F_{all}, its size is strictly increasing and bounded by $|F_{\text{all}}|$: $|\mathcal{P}_{i+1}| > |\mathcal{P}_i|$ and $|\mathcal{P}_i| \leq |F_{\text{all}}|$. Thus, the sequence of iterations must be finite. The process terminates when either $SE_i = \emptyset$, resulting in **Success** ($R(S_i) = C_i$), or when the repair tool returns **Failure** ($R(S_i) = \text{Unsatisfiable}$). □

3.6 Case Study: Integrating ISR with Three Different Types of Repair Tools

In this section, we demonstrate the effectiveness of ISR with existing repair tools to generate configurations without undesired side effects in the following OSPF and BGP repair scenarios.

Table 3.1: Comparison of OSPF repair results.

	NetComplete	**S^{2}Sim**	**NetBuddy**
Number of refinement iteration	6	3	2
Average number of specifications added in refinement	20.0	5.5	11.0
Number of conflict resolution step	2	2	1
Repair correctness	guaranteed	guaranteed	not guaranteed

We fully implemented ISR in 10k lines of C++ code and integrated it with NetComplete (SMT-based) [99], S^{2}Sim (Simulation/SMT-based) [29], and NetBuddy (LLM-based) [94,116], using their publicly available implementations [125,126]. We reproduce S^{2}Sim by following the repair algorithms in the original paper. The implementation of ISR will be made publicly available upon acceptance.

In the OSPF repair (§3.6.1), we use NetComplete and S^{2}Sim by fixing 90% of the link costs to mitigate side effects, thereby limiting the tool to modify only the remaining 10%. If the repair was unsat, we decreased the percentage of fixed links to 85%. In BGP repair (§3.6.2), we use the tool with all configurations made symbolic. As an LLM for NetBuddy, we locally run the gpt-oss:120b model. We run the experiments on a single core in a computer with an Apple M1 Ultra (128 GB RAM).

3.6.1 OSPF Configuration Repair (Repair for Shortest-Path Routing)

We first show the effectiveness of ISR for forwarding path changes in OSPF networks. We use the real topology, Arnes (34 routers and 94 links), from the Topology Zoo dataset [127]. On the topology, we assign random costs to each link and compute all the shortest paths between the nodes.

Repair scenario. We consider a repair scenario where the goal is to change a shortest path between two nodes to a suboptimal (i.e., non-shortest) path, reflecting a realistic requirement often encountered when switching to a backup path. For this scenario, we first randomly select a source-destination pair and compute a valid suboptimal path between them. We then use this path change intent as the initial repair specification and task ISR with guiding the three repair tools to find a link cost update set that satisfies this specification without introducing undesired side effects. Note that when the specification for the change intent is in conflict with the specifications to prevent side effects, we prefer the one for the change intent for conflict resolution.

Results. ISR successfully guided these tools to generate configurations satisfying the change intent without undesired side effects. Table 3.1 shows the metrics during the refinement, indicating differences in the refinement effort across the tools. NetComplete required the most refinement, with 6 iterations and an average of 20.0 specifications added to prevent side effects, followed by S^{2}Sim with 3 iterations, 5.5 specifications. NetBuddy required 2 iterations with 11 additional specifications.

These differences stem from their underlying mechanisms of configuration parameter

search employed by each repair tool. NetComplete and S^{2}Sim, relying on an SMT-solver, generate any link cost changes that satisfy the specifications, which can affect unspecified paths. In contrast, NetBuddy (LLM-based) appears to have a reasoning power that generates link cost changes while minimizing the impact on other forwarding paths. However, NetBuddy sometimes returned link cost changes that did not satisfy the repair specifications (Repair correctness). Ensuring the correctness of LLM-based repair is beyond the scope of this paper, so we discuss these cases in §3.9. In contrast, NetComplete and S^{2}Sim are guaranteed to generate configurations that satisfy the specifications, thus ensuring the refinement process with it will always converge.

In summary, ISR guided the three different types of repair tools to generate link cost updates that satisfy the path change intent without introducing undesired side effects in the OSPF network.

3.6.2 BGP Configuration Repair (Repair for Policy-based Routing)

Next, we demonstrate the effectiveness of ISR for BGP configuration repair. Since S^{2}Sim completes the repair without side effects, we report the results of NetComplete and NetBuddy. We use the runtime configuration change and its topology (Figure 3.5) of the repair scenario in [116] with an additional prefix (20.0.0.0/16 on Provider 1). Prior to the repair, Customer 1 directly reaches Provider 1. To reach Customer 2, Customer 1 has two paths: (1) primary path ($Customer1 \rightarrow Provider1 \rightarrow Provider2 \rightarrow Customer2$) and (2) backup path ($Customer1 \rightarrow Provider2 \rightarrow Customer2$). Customer 1 enforces this path preference by reducing the local preference of routes advertised by Provider 2.

Repair scenario. AS 100 operators plan to change the primary path via Provider 2. We task NetComplete and NetBuddy with generating BGP configurations to achieve this path change. We provide the four routers' configurations and the two specifications for prefix 200.0.0.0/16: (1) the primary path (200.0.0.0/16, $Customer1 \rightarrow Provider2 \rightarrow Customer2$) and (2) the backup path (200.0.0.0/16, $Customer1 \rightarrow Provider1 \rightarrow Provider2 \rightarrow Customer2$).

Repair by NetComplete with ISR. Given the configurations and specifications, NetComplete generates two route maps for Customer 1. The first route map permits all routes advertised from Provider 2 with default local preference (100), so the path order in the specifications is satisfied purely by AS-path length. The second route map, however, includes a condition that denies all prefixes advertised from Provider 1 except 200.0.0.0/16. Given the repaired configurations, ISR (1) identifies the reachability lost for prefix 20.0.0.0/16 on Provider 1 from Customer 1 as a side effect and (2) refines the specifications to prevent the side effect by adding the new path requirement (20.0.0.0/16, $Customer1 \rightarrow Provider1$).

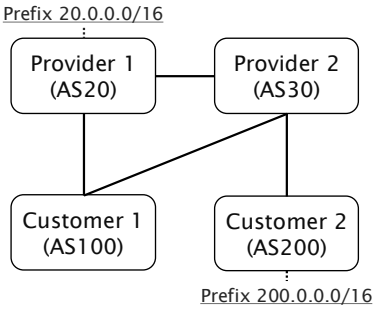


Figure 3.5: A BGP network for repair.

Given the new specifications, NetComplete generates two route maps identical to the previous ones, except for adding the 20.0.0.0/16 permit. ISR confirms that the repaired configurations satisfy the specifications and do not introduce any side effects.

Repair by NetBuddy with ISR. Given the configurations and specifications, NetBuddy

generates route maps on Customer 1 that assign a local preference 100 to routes from Provider 1 and 200 to routes from Provider 2. Consequently, the prefix 20.0.0.0/16 is unintentionally routed via Provider 2 due to the higher local preference. Given the configurations after repair, ISR (1) identifies this path change for prefix 20.0.0.0/16 as a side effect and (2) refines the specifications by adding the new path requirement (20.0.0.0/16, $Customer1 \rightarrow Provider1$).

Given the new specifications, NetBuddy generates new route maps that apply this local preference assignment (100 for Provider 1 and 200 for Provider 2) only to the prefix 200.0.0.0/16, thereby ensuring that the route for 20.0.0.0/16 remains unchanged. ISR confirms that the configurations satisfy the specifications and introduce no further side effects.

In summary, ISR correctly guides the repair tools to generate configurations that satisfy the path change intent without introducing undesired side effects in the BGP network.

3.7 Scalability Evaluation

Next, we show that ISR performs side effect diagnosis and specification refinement within a reasonable time on real-world networks of various sizes and with varying numbers of side effects (§3.7.1 and §3.7.2). Furthermore, our specification minimization technique effectively reduces the number of specifications for the next repair by 2x on average (§3.7.3). We run all the experiments 100 times on a single core in a computer equipped with an Apple M2 Max processor with 32 GB RAM.

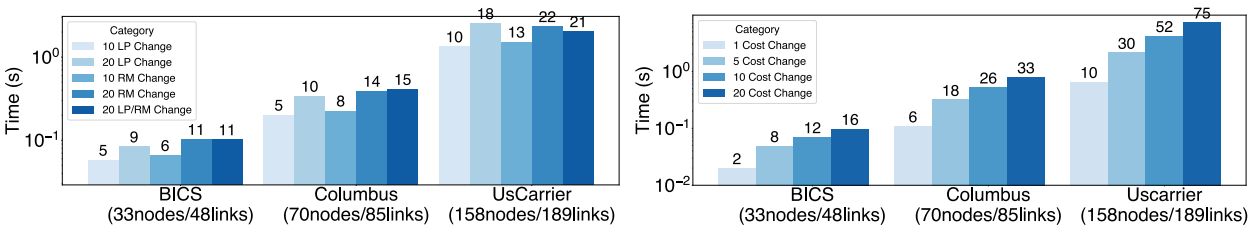
3.7.1 How Scalable is ISR for Repair with Various Network Sizes?

We first evaluate the scalability of ISR for configuration repair across various network sizes.

Topology and configurations. We pick up three real network topologies from Topology Zoo dataset [127]: BICS (33 nodes and 48 links), Columbus (70 nodes and 85 links), UsCarrier (158 nodes and 189 links), which include both commercial and educational networks. For each topology, we use synthetic BGP and OSPF configurations, which are publicly opened [128].

Configuration changes as repair. On these datasets, we evaluate the processing time of side effect diagnosis and specification refinement for an iteration by randomly changing the router configurations to emulate configuration repair. In the BGP scenarios, we introduce two types of changes as common BGP errors to induce side effects [122]: local preference changes and route denial from peers. For local preference, we randomly select a router and one of its BGP peers, increasing the local preference for routes received from that peer from the default 100 to 150. For route denial, we install a route map on a randomly selected router to deny all incoming routes from one of its peers. In the OSPF scenarios, we randomly select link costs between routers and increase them by random values. To evaluate the performance under worst-case conditions in the BGP and OSPF configuration changes, ISR classifies every detected path change as a side effect.

Results. Figure 3.6 gives the average time required for a single cycle of side effect diagnosis and specification refinement across each configuration change. The processing time generally scales with the network size and the complexity of the change. For both BGP and OSPF configuration changes on the largest networks, ISR performs the refinement within seconds. In addition, we observe that the processing time slightly increases with an increase



(a) BGP configuration changes.

(b) OSPF configuration changes.

Figure 3.6: Processing time for side effect diagnosis and specification refinement per iteration. Numbers on bars indicate the average number of side effects per prefix. LP and RM denote Local Preference and Route Map.

in the number of side effects from more configuration changes. Note that OSPF configuration changes are more prone to introducing side effects than BGP, because a single link cost change can potentially impact the forwarding paths for all prefixes. In summary, ISR can execute individual refinement cycles in a reasonable time to perform the iterative specification refinement in real-world settings.

3.7.2 How Scalable is ISR for Repair with Various Numbers of Side Effects?

Second, we evaluate how ISR scales with an increasing number of side effects. Note that while a large number of side effects evaluated here are unlikely to occur in practice, we conduct this experiment to show the processing time trend of ISR as the number of side effects increases.

Topology and configurations. To induce a large number of side effects, we use the largest topology from Topology Zoo [127]: Kdl (754 nodes/899 edges). On this topology, we synthesize BGP configurations where each router is assigned a unique AS number and peers with its adjacent nodes. Each router advertises a unique IP prefix and reaches other routers via one of the shortest paths.

Configuration changes as repair. To produce side effects, we randomly inject the local preference and route map changes, following the method in §3.7.1. For worst-case, ISR is configured to classify every best path change as a side effect.

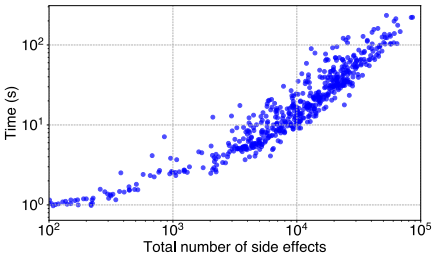


Figure 3.7: Time for specification refinement with side effect diagnosis with varying numbers of side effects.

Result. Figure 3.7 shows the processing time trend of performing a single specification refinement iteration. We observe a strong positive correlation between the two variables, confirming that the processing time increases as the number of side effects grows. Crucially, the linear-like trend on the log-log scale indicates that this relationship is polynomial, not exponential, which demonstrates the computational feasibility of our approach. For instance, scenarios involving approximately 10^3 side effects are processed in a few seconds, while larger cases with 10^4 side effects complete within 10 to 20 seconds. Even in extreme scenarios with nearly 10^5 side effects, the processing time remains manageable (typically under 150 seconds),

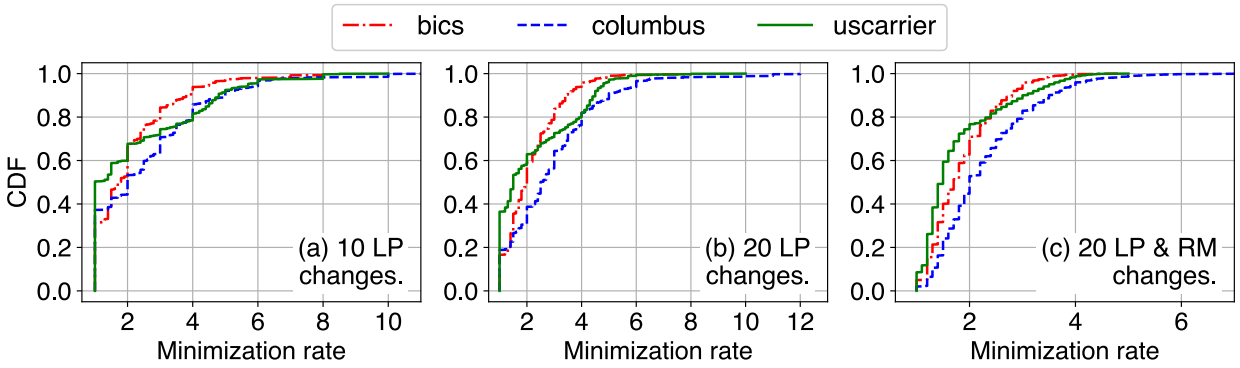


Figure 3.8: Specification minimization for BGP configuration changes.

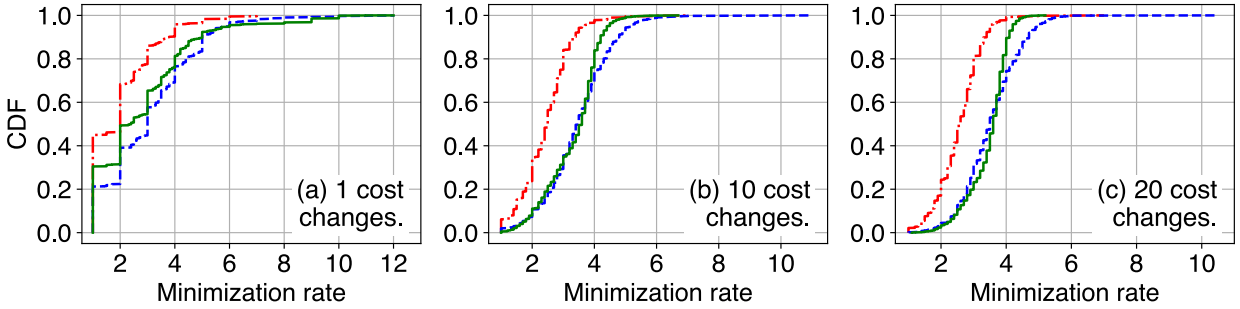


Figure 3.9: Specification minimization for OSPF configuration changes.

confirming that our technique is sufficiently scalable for a large number of side effects.

3.7.3 How Effective is the Specification Minimization?

Finally, we show how ISR effectively minimizes the specifications added to the next repair.

Topology and configurations. To measure the specification minimization rate, we use the same network topologies and configurations in §3.7.1. For BGP configurations, we pick up (1) local preference changes and (2) both local preference and route map changes. We share the results of the route map changes in §A.4. For OSPF configurations, we pick up 1, 10, and 20 cost changes.

BGP Result. Figure 3.8 shows the cumulative distribution functions (CDFs) of the specification minimization for the BGP configuration changes. Our technique consistently achieves a reduction (rate $>$ 1.0), with a median rate ranging from 1.2x to 2.8x. We observe that this rate is topology-dependent; UsCarrier (green line) consistently underperforms the other topologies. Furthermore, the change type impacts performance: while the local preference changes from 10 (a) to 20 (b) had little effect, adding route map changes (c) degraded the minimization effectiveness for all topologies.

OSPF Result. Figure 3.9 presents the corresponding results for the OSPF configuration changes. The specification minimization is highly effective, with median reduction rates clustering between 2.0x and 3.0x across all the cost changes. The main reason for this high effectiveness is that OSPF selects the shortest paths using uniform link costs for all prefixes. This path selection introduces greater path compositionality, allowing for more effective minimization than the BGP results.

3.8 Limitation

Supported network protocol and layer. ISR currently supports BGP and OSPF in layer-3 networks. Supporting other protocols (e.g., MPLS, RIP, and Segment Routing) and layers (e.g., layer-2) requires extending ISR to model their semantics, which is an important direction for future work.

Link failure. To consider multiple link failures, ISR simulates route computation for each combination of link failures. One extension of ISR to efficiently consider the link failure scenarios (known as *k-link failure*) leverages the graph-based techniques [29,35,37].

Dynamic policy. Resource-aware intent specifications incorporate resource constraints to satisfy dynamic network properties such as bandwidth and QoS [129–131]. As with existing repair tools, ISR currently targets static forwarding path change intents to prevent side effects.

3.9 Discussion

Uncertainty of Configuration Repair with LLM. Leveraging LLMs to infer configuration parameters that satisfy new network change intents [94,100,116,132] has the potential to aid in solving computationally expensive synthesis and repair problems [133]. An inevitable challenge in LLM-based configuration repair is that it is not formally guaranteed to generate router configurations satisfying the given specifications. This stems from the fact that LLM reasoning relies on statistical pattern correlation rather than formal logic, which can produce plausible but factually incorrect outputs [95,103,134–136].

In our OSPF repair experiment (§3.6.1), we observed that NetBuddy, an LLM-based repair tool, sometimes returned link cost changes that failed to satisfy the given specifications. Furthermore, it occasionally generated cost changes for non-existent links. We identified these invalid cost changes by comparing the simulation results of the route computations before and after the repair. Upon detecting these situations, we restarted the repair experiment from scratch. To verify if this was specific to the model we used for demonstration (gpt-oss:120b), we tested other LLMs (Llama 3.1 70B, Code Llama 6B, and gpt-oss 20B) under the same conditions; however, we observed the same phenomenon. Therefore, tools such as network verifier [33,52,78,123,124] and ISR are essential to validate the configurations generated by LLM-based repair tools. Note that the following observation and discussion do not indicate a limitation of the NetBuddy implementation [126] and paper [94,116]; rather, we highlight the inherent difficulty of using LLMs to correctly infer configuration parameters that satisfy operator's network change intents.

Repair approaches based on formal methods, such as NetComplete [99], are not subject to this uncertainty and are guaranteed to generate link costs that satisfy given repair specifications (if a solution exists) [29,97–99,102,104–106]. However, these approaches suffer from scalability problems, often requiring several hours or more to perform configuration synthesis/repair for networks with several hundred network devices. Developing a configuration repair approach that achieves both scalability and correctness remains an important direction for future research.

Equal-cost multiple path. ISR can consider multiple best paths with equal costs (e.g.,

ECMP) during the side effect diagnosis by simulating and comparing the complete set of forwarding paths before and after repair. For example, if the reachability of one of the equal-cost best paths is lost after the repair, ISR identifies this as a path change and can determine whether it is a side effect.

Non-convergence. If routes do not converge in configurations after repair, the route computation simulation also does not converge in ISR. In this scenario, ISR can identify which configuration changes after repair introduce the problem by terminating the simulation at a predefined step.

Incomplete/Ambiguous intents. Recent works [103,121] attempt to reduce the incompleteness and ambiguity of operators' change intents for configuration synthesis/repair. SEA [121] assists operators in diagnosing side effects and their sources, guiding manual updates to repair specifications. In contrast, we propose a general framework for diverse repair tools that automatically compensates for the incompleteness of change intents. By incorporating these tools into an iterative specification refinement (§3.3.2), we guide them to satisfy change intents while preventing undesired side effects. Clarify [103] resolves the ambiguity of operator's intents for LLM-based ACL and route map updates by interactively helping operators to select the desired behavior from differential examples. Our framework targets the incompleteness of change intents regarding network-wide forwarding paths. By automatically preserving these paths from undesired side effects, we enhance various repair tools to ensure network integrity beyond the local correctness of individual configurations.

Specification for management objectives. In addition to satisfying forwarding path requirements, operators often have "management objectives" for repair, such as minimizing the number of network devices or configuration lines changed. To ensure these management objectives, several repair tools incorporate them into their repair process [97–99,101,105]. For instance, AED [98] introduces a high-level language to describe these objectives and translates them into optimization problems with the path requirements in repair (using soft constraints in MaxSMT problem). These approaches are complementary to ISR, which focuses on preventing side effects induced by new path requirements rather than optimizing the manageability of the configuration changes.

Configuration synthesis. Configuration synthesis generates router configurations to satisfy given forwarding requirements *from scratch* [99,101,103–106,137–139], which can be applicable for repair to meet new change intents. ISR remains effective in this repair scenario by iteratively refining specifications based on the synthesized and current configurations and the requirements.

Reconfiguration ordering. The order in which the configurations after repair are deployed to networks is critical, as an incorrect ordering can induce transient failures during convergence. Several works address this problem by computing a correct ordering of the deployment to prevent such failures [123,140–142]. ISR is complementary to these systems: it guarantees that the configurations after repair do not introduce undesired side effects, and these ordering systems can then be used to ensure the transition to that configuration is also safe.

Control plane simulation. Control plane simulation of ISR is highly inspired by the existing simulation techniques [32–37,78,81]. We can extend the underlying simulation of ISR with emerging techniques, especially for high scalability (e.g., [81]) or k-link failures (e.g., [35,37]).

3.10 Summary

The risk of side effects in configuration repair hinders the practical adaptation, as forwarding paths not described in specifications can change without the operator's awareness. We proposed ISR, an iterative specification refinement framework that externally guides repair tools to generate configurations satisfying network change intents without undesired side effects. Our evaluation demonstrated ISR's effectiveness, showing that it successfully guided three different repair tools to satisfy change intents without undesired side effects in OSPF and BGP networks. Furthermore, we confirmed ISR's scalability, completing a single refinement within seconds on real network topologies with up to 200 routers.

Chapter 4

Symbolic Extraction of Packet Forwarding Behaviors with Partial Queries

In this chapter, we introduce a symbolic extraction approach of packet forwarding behaviors without complete specifications.

4.1 Introduction

Network troubleshooting requires operators to identify the causes and impacts of the failures by investigating how the networks forward packets. Operators start troubleshooting from notification of network failures (e.g., alerts from monitoring systems and/or their customers). It is generally hard to pinpoint the causes and impacts of failures only with the notification. Therefore, operators (1) continually probe networks using traditional tools (e.g., ping and traceroute) or sophisticated ones [33,62,143] and (2) reason how packets are forwarded on the networks step-by-step.

Investigating how networks forward packets (i.e., packet forwarding behavior) has become complex due to various routing protocols and heterogeneous network devices. Today's networks are composed of not only traditional protocols (e.g., OSPF and ISIS) but also fine-grained access control, traffic engineering, and sometimes SDN-like controls [5–7] on network devices of multiple vendors [8,9]. Reasoning the forwarding behaviors determined by their interactions is highly difficult, even though it is inevitable to identify the causes and impacts of network failures. Thus, operators spend hours to days identifying the causes and impacts [55,143–145].

From the complex packet forwarding behaviors, extracting specific behaviors related to network failures would help operators identify the causes and impacts of the failures. Imagine an operator who tries to investigate forwarding behaviors destined to a suspicious prefix related to a failure. If the operator can obtain all the behaviors (packet headers and their forwarding paths) destined to the prefix at first, she could concentrate on checking the faulty behaviors to identify the failure causes and impacts. However, probing with existing tools is not capable of such comprehensive extraction because it depends on concrete parameters

in a header (e.g., source/destination addresses and ports) and gives operators a forwarding behavior matching the parameters in a single probe.

To comprehensively investigate packet forwarding behaviors, data plane verification has explored the static analysis of forwarding rules collected from routers and switches (e.g., FIBs and ACLs) [26,43–48,51,52,55,58,93,146]. This approach (1) models packet forwarding behaviors as the combination of packet headers and their forwarding paths and (2) represents all the behaviors on a single logical model. Thanks to the model, this approach simultaneously verifies forwarding properties of multiple forwarding behaviors with *symbolic* packet headers (e.g., destination prefix 1.0.0.0/24 and arbitrary port numbers) and forwarding paths (e.g., any paths from a node to another node), unlike the concrete parameters in traditional probing. For instance, it verifies if the following specification is satisfied: "any packets with destination prefix 1.0.0.0/24 reach from node A to B."

While the data plane verification approach is beneficial for network operators to comprehensively investigate packet forwarding behaviors, this approach is orthogonal to *extract* forwarding behaviors matching packet headers and forwarding paths related to network failures for troubleshooting. The verification approaches focus on answering the binary question: *verifying* if given specifications are satisfied or not on a model. The underlying data structures and algorithms are originally designed to return either 'yes' or 'no' with a counterexample to the given specifications. To help operators identify the causes and impacts of network failures in troubleshooting, we need an approach that enables the operators to (1) symbolically specify packet headers and/or forwarding paths related to the failures and (2) obtain all forwarding behaviors matching the symbolic values.

To show the effectiveness of the symbolic extraction approach for troubleshooting, we propose Lupe, a system that extracts forwarding behaviors matching given symbolic packet headers and/or forwarding paths from all possible behaviors. Using Lupe, operators can ask questions that active probing and data plane verification do not directly answer to Lupe such that (1) "Show all forwarding paths of packet headers destined to prefix 1.0.0.0/16" and (2) "Show any types of (symbolic) packet headers forwarded through a flapped link." We design a query language for operators to easily ask such questions to Lupe (§4.5.1). By querying Lupe with symbolic headers or paths related to network failures, operators can obtain and investigate all forwarding behaviors matching the symbolic values to identify the causes and impacts of the failures.

Answering such questions poses two challenges: (1) data models representing all possible forwarding behaviors to effectively extract the behaviors matching symbolic headers or paths, and (2) search algorithms to extract the matching behaviors from all the behaviors. For the first challenge, we compute all possible behaviors in two distinct spaces: header and path space. Each space symbolically represents all possible headers or forwarding paths. Creating such distinct spaces enables efficient extraction of the behaviors matching symbolic headers or paths because we only need to search either space corresponding to the query type with space-specific pruning. We design a scalable algorithm to construct the two spaces based on a lattice [147] (header space) and a directed graph (path space) (§4.4). The key idea is to compute header spaces for disjoint prefix groups and merge them into a single space. This prefix-based grouping distributes the overall complexity of the header space computation across disjoint prefix groups, which scales the header and path space creation of Lupe for large-scale production datacenter networks and complex ACLs (§4.6.1).

Table 4.1: Comparison of three approaches. Lupe enables network operators to extract all forwarding behaviors matching symbolic packet headers or forwarding paths by a single query.

	Input	Search space	Output	What operator obtain
Active probing	*Concrete* packet header and source/destination nodes to probe	Actual network	*Concrete* packet header, its forwarding path, and its forwarding properties (e.g., reachability, latency)	*Extract* a forwarding behavior of the packet
Data plane verification	Specification: Triplet of *symbolic* header, *symbolic* forwarding path and property	Model from forwarding rules (e.g., FIB, ACL)	Valid or Invalid about input specification	Know if specification is satisfied on *all possible behaviors*
Lupe	Query: *Symbolic* header or *Symbolic* forwarding path	Model from forwarding rules (e.g., FIB, ACL)	*Symbolic* packet headers, their forwarding paths, and their forwarding properties matching the input query	*Extract* forwarding behaviors matching the query from *all possible behaviors*

For the second challenge, we design two symbolic search algorithms to extract forwarding behaviors matching the symbolic headers or paths from the two spaces (§4.5). The search spaces of possible behaviors could be huge in large-scale networks, so the naive search is inefficient in supporting realtime troubleshooting. Both algorithms reduce the search spaces to only those subspaces relevant to the queries by treating the given headers and paths as constraints in the search processes. Thanks to the search space reduction, Lupe can handle various types of queries to extract forwarding behaviors in realtime troubleshooting (§4.6.2).

We evaluate Lupe using (1) the various sizes of synthesized WANs/LANs/ACLs and real production datacenter networks and (2) case studies in real campus and production datacenter networks (§4.6). We first show that Lupe extracts forwarding behaviors matching given symbolic headers or paths from all possible behaviors in 15 seconds, even in the large-scale data center network, with 5x-129x improvement over techniques applying existing algorithms for data plane verification. Second, we demonstrate how Lupe helps operators identify the causes and impacts of real failures (performance degradation and silent packet drop) in the two networks through the case studies.

4.2 Related work and Motivation

There are two orthogonal approaches to investigate packet forwarding behaviors in troubleshooting: active probing and data plane verification. We introduce both approaches with Table 4.1 and motivate the importance of the symbolic extraction of packet forwarding behaviors for troubleshooting.

4.2.1 Active probing

Probing networks with testing tools (e.g., ping/traceroute) is a common approach to investigate current packet forwarding behaviors and obtain their information (first row in Table 4.1). Given a packet header describing the fields required for forwarding, network operators send the packet from source to destination nodes on networks. By observing the fate of the packet, operators gain insight into the forwarding properties of the packet (e.g., reachability and forwarding paths).

One of the benefits of the probing in troubleshooting is that operators can collect

forwarding behaviors only with partial information observed in troubleshooting. In typical troubleshooting, operators repeat the following two processes: (1) observe the network partially (e.g., using ping and traceroute) and (2) speculate on what happens based on the packet headers and forwarding paths of the observation. Through the processes with partial headers and paths, operators reason how packets are forwarded on the network step by step to identify the causes and impacts of failures.

Previous studies have explored automatic approaches to find the causes of network failures or explain how networks behave by leveraging data collected through probing. Fault localization techniques (1) create statistical/probabilistic models based on the probed data and (2) infer network failures and their causes that could not be identified without the probing (e.g., ASIC and driver bugs) in large-scale datacenter networks [62,66,69,143,148,149]. Net2Text [145], Anime [117], and network provenance techniques [150–152] collect probed forwarding stats per forwarding paths and summarize current forwarding behaviors for operators to easily understand them.

4.2.2 Data plane verification

Unlike active probing that sends packets on networks, data plane verification statically analyzes packet forwarding rules (e.g., FIBs and ACLs) collected from routers and switches (second row in Table 4.1) [26,43–45,47,48,51,55,58,93,146]. Using forwarding rules and network topology, it represents all possible forwarding behaviors (i.e., all the combinations of forwarding paths and the set of packet headers on the paths) on a logical model. The model compactly represents the set of packet headers on each forwarding path by symbolically representing the header set.

The unique benefit of this approach for troubleshooting is to accept symbolic expressions of packet headers and forwarding paths to investigate packet forwarding behaviors. For instance, this approach can return yes or no to the following specification at once: "any packets with destination prefix 1.0.0.0/16 reach from node A to B." If the specification is violated, it also returns a counterexample (e.g., a concrete packet header violating the specification). Unlike active probing, this approach can comprehensively investigate forwarding behaviors matching symbolic headers and paths at once. Since this approach creates the model based on forwarding rules and topology, it is mainly used for identifying faulty behaviors due to forwarding rules that are contrary to operator's expectation (e.g., misconfiguration of routing protocols and ACLs [8,9,33,143,153]).

4.2.3 Motivation: Symbolic extraction of packet forwarding behavior

Despite their unique benefits, there is no approach that simultaneously achieves both benefits: (1) extraction of packet forwarding behaviors (active probing) and (2) symbolic expression of packet headers and forwarding paths to comprehensively investigate forwarding behaviors (data plane verification). Achieving both benefits helps network operators obtain forwarding behaviors related to failures using the symbolic expression without the need for continuous probing, allowing them to concentrate on investigating the causes and impacts of the failures.

We highlight the importance of the symbolic extraction approach with a simple troubleshooting example in Figure 4.1. Here, operators receive an alert from Site A that the

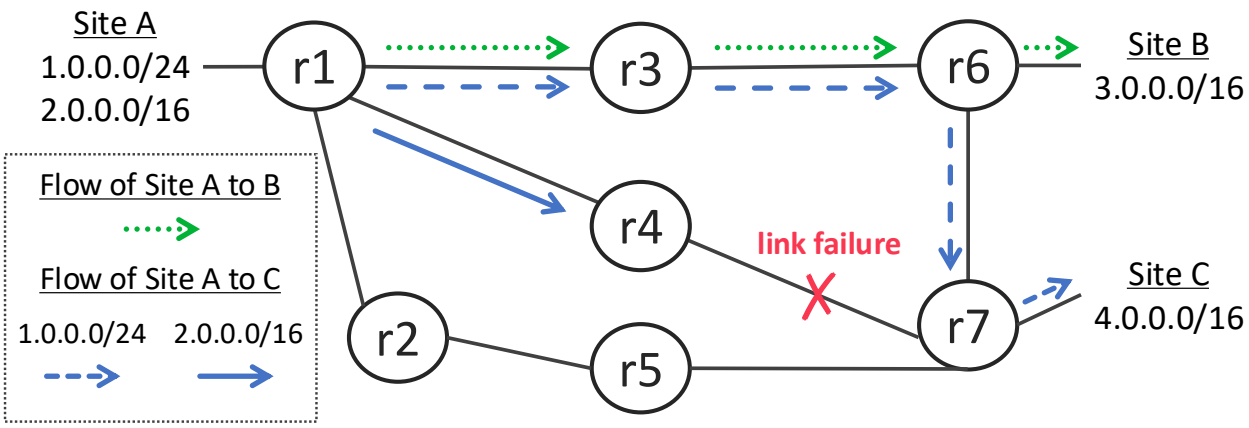


Figure 4.1: An example scenario of troubleshooting.

source prefix 2.0.0.0/16 can not reach the destination prefix 4.0.0.0/16 in Site C. In this troubleshooting, answering the following two questions is helpful for operators to identify the causes and impacts of the reachability failure: (1) "Show all (symbolic) packet headers that can not reach from Site A to Site C" and (2) "Show all forwarding paths reachable from Site A to Site C." Active probing depends on concrete packet headers or forwarding paths in a probe, so it does not answer such questions. Data plane verification focuses on verifying if given specifications are satisfied (i.e., answering binary questions) and does not extract forwarding behaviors matching the two questions. How to achieve the symbolic extraction approach for troubleshooting is still an open problem.

4.3 Approach

We propose Lupe, a symbolic extraction of packet forwarding behaviors that simultaneously achieves both benefits of active probing and data plane verification (third row in Table 4.1). Out of forwarding rules and network topology, Lupe extracts forwarding behaviors matching given symbolic packet headers or (and) forwarding paths. By querying Lupe with symbolic headers or paths related to network failures, operators can obtain the forwarding behaviors matching the query to identify the causes and impacts of the failures.

Workflow with example: Operators can query forwarding behaviors from Lupe aligned with two types of queries—header query and path query—that accept symbolic descriptions in our query language (the syntax is described in §4.5.1). A header query is symbolically defined on header fields such as prefixes for source/destination addresses and ranges or sets for port/protocol numbers. For instance, any packets with destination addresses 4.0.0.0/16 and destination port range 200-250 are described as [DstIP:4.0.0.0/16, DstPort:[200,250]]. Unspecified fields in header queries are treated as wildcards. A path query is defined by a complete or partial sequence of network device identifiers (e.g., name and router-id) as directed paths. In the sequence, hop counts of each hop are quantified by a specific hop (path$\{N\}$), a range of hops (path$\{N_1, N_2\}$), or zero or more hops (path*). For instance, any paths from router ⓡ1 to ⓡ7 through ⓡ3 are described as [r1 .* r3 .* r7]. The ".*" means that any identifier (".") occurs zero or more times. Lupe answers the queries by extracting all

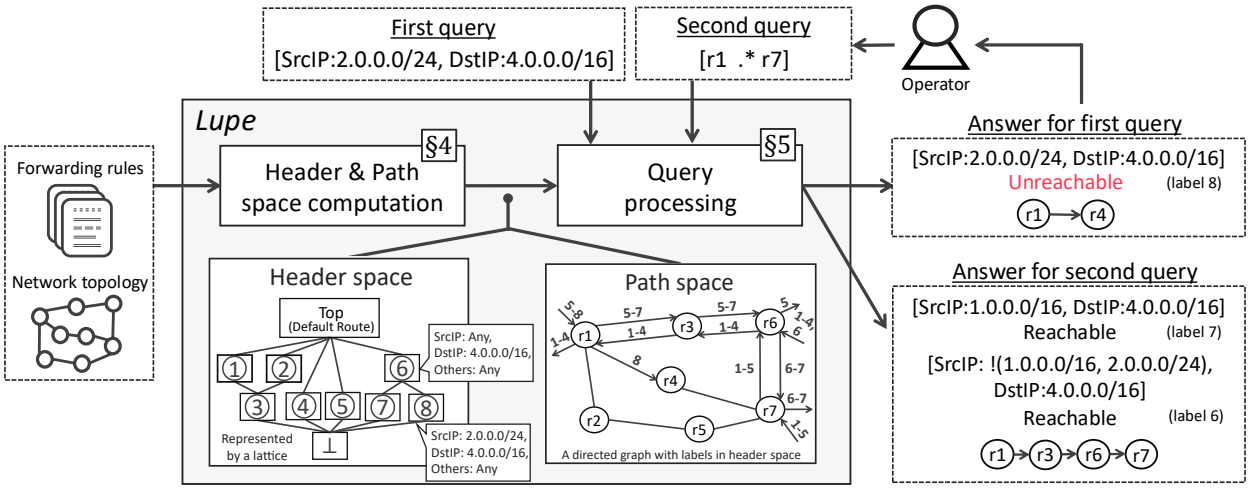


Figure 4.2: The overall workflow of Lupe and key components.

forwarding behaviors matching the queries. Each forwarding behavior is represented by a triplet of header constraint, forwarding path(s), and its forwarding properties. Lupe supports several practical properties, including reachability, forwarding loop, blackhole, isolation, waypoint, and hop limit, by analyzing forwarding paths matching given queries (§4.5.1).

Figure 4.2 illustrates how Lupe helps operators identify the failure cause and impacts in the example of Figure 4.1. Lupe leverages partial header and path information that the operator observes and uses as clues in troubleshooting to extract forwarding behaviors. Since the operator first investigates the cause of the reachability failure for the source prefix 2.0.0.0/16, she gives the header query specifying the prefix (first query in Figure 4.2) to Lupe. From the output, she finds that the packets destined to the prefix 4.0.0.0/16 are dropped at router (r4). She checks the router (r4) and detects a link failure between routers (r4) and (r7). Second, she tries to check if the link failure impacts other traffic from Site A to C. To this end, she gives the path query that specifies any traffic from router (r1) to (r7) (second query in Figure 4.2) to Lupe. She then confirms that traffic from the prefix 1.0.0.0/24 in Site A to the prefix 4.0.0.0/16 in Site C is routed through routers (r3) and (r6) from the output.

Technical challenge: The challenge in designing Lupe is to extract forwarding behaviors that match *either* symbolic headers *or* symbolic paths from all possible behaviors within realtime constraints of troubleshooting (e.g., SLA [154–156]). While the symbolic analysis in data plane verification can be a key enabler, simply applying the existing algorithms and data structures does not fit for *extracting* and *outputting* the behaviors for given symbolic headers or paths.

The state-of-the-art data plane verifiers represent header constraints on each forwarding path using Binary Decision Diagrams (BDDs) [92] and incrementally verify forwarding properties for given data plane changes (e.g., link down) [43,44,48,51,52,146]. BDD is powerful in answering if given headers are forwarded on given paths, as it compactly represents a Boolean function. However, the BDD-based approach is unsuitable for extracting forwarding behaviors matching given symbolic headers. Since this approach represents header constraints per forwarding path, it requires the intersection between a given header constraint and the constraints on all the forwarding paths for the extraction, which is inefficient as the number of forwarding paths increases in large-scale networks. Other than BDD, existing verifiers

represent possible header constraints on all forwarding paths using a single data structure (as header space), such as a multidimensional trie [45] and lattice [47,89] with a graph (as path space). These techniques have scalability issues for representing the header constraints. In computing header space, they naively check intersection and inclusion relations between header constraints in forwarding rules. In the worst case, they need $\mathcal{O}(N^2)$ relation checks where N is the total number of possible header constraints. We confirm that the computation time of these techniques dramatically increases as the number of these relations increases in large-scale networks (e.g., route summarization [10,157]) and complex ACLs with multiple header fields, which is not suited to the realtime constraints of troubleshooting (§4.6.1).

Contribution: To efficiently extract and output forwarding behaviors matching symbolic headers and/or forwarding paths, we design Lupe with two techniques. First, we design a scalable algorithm to symbolically compute all possible forwarding behaviors in distinct header and path spaces (§4.4). We reduce the total number of intersection and inclusion relation checks between header constraints by computing header spaces for disjoint prefix groups and merging them into a single header space. The prefix-based grouping distributes the relation checks for all the header constraints across each disjoint group, thereby reducing the overall computational complexity (the complexity analysis in §4.4). This technique makes the header and path space computation scalable for large production datacenter networks and complex ACLs with multiple header fields (§4.6.1). Second, we design two search algorithms on each space to extract forwarding behaviors with either symbolic headers or paths (§4.5). Both algorithms treat the symbolic headers and paths as search constraints and reduce the search spaces to the subspaces relevant to the queries by checking if the constraints are satisfied in every search step. The search space reduction enables Lupe to answer various queries in realtime troubleshooting (§4.6.2). With this capability, we demonstrate that Lupe helps operators identify the causes and impacts of real network failures in campus and datacenter networks (§4.6.3 and §4.6.4).

4.4 Header and Path Space Computation for Extraction

To extract forwarding behaviors matching symbolic headers or paths, Lupe first derives all possible behaviors from forwarding rules (e.g., FIBs, ACLs) and network topology. As illustrated in Figure 4.2, these behaviors are represented in two distinct spaces: *header space* that represents all possible headers and *path space* that represents forwarding paths of these headers. The distinct spaces enable Lupe to extract the behaviors matching symbolic headers or paths by exploring either space corresponding to queries (§4.5). In this section, we describe how Lupe computes the two spaces.

4.4.1 Header space computation

To symbolically compute all possible headers in a header space, Lupe analyzes all intersection and inclusion relations between header constraints in forwarding rules. As an example, let us consider how Lupe processes two rules with the same destination prefix 1.0.0.0/16 and different destination port ranges [2000:2500] and [2250:2750]. To distinguish packet headers within the common port range of the two rules [2250:2500] and other disjoint ranges

[2000:2250] and [2500:2750], Lupe (1) identifies the common range by intersecting the two rules and (2) creates a new symbolic header with the prefix 1.0.0.0/16 and the common range [2250:2500]. Formally, we define the intersection and inclusion relations as follows:

Definition 2. (Intersection between packet headers) Let h_1, h_2 be packet headers, and the i th field of the header h_1 be represented like h_{1i}. h_1 and h_2 are intersected if $h_{1i} \cap h_{2i} \neq \emptyset$ for any i such that $1 \leq i \leq n$ where n is an index for the last header field.

Definition 3. (Inclusion between packet headers) Let h_1, h_2 be packet headers. h_1 includes h_2 if $h_{1i} \supseteq h_{2i}$ for any i such that $1 \leq i \leq n$.

To analyze these relations, Lupe leverages a lattice [147] because it has a useful mathematical structure for the analysis. The important property is that every pair of nodes in a lattice has a least upper bound and a greatest lower bound. In our use case, it ensures that a header constraint of any parent node includes the constraints of their children (e.g., the constraint of ⑥ includes ⑦'s and ⑧'s on header space in Figure 4.2), and constraints of children with two parents are the common part of the parents' constraints (e.g., ① ∩ ② = ③ in Figure 4.2). Leveraging this property, Lupe computes all possible headers by creating header space on a lattice (the proof is in §B.1).

We design a scalable algorithm that reduces the computational complexity of the header space computation on a lattice by pruning redundant checks of intersection and inclusion relations. For pruning, we have three steps: (1) identify groups of disjoint destination prefixes in forwarding rules, (2) create header spaces for each disjoint group, and (3) merge them into a single header space. The prefix-grouping distributes the complexity of relation checks across each disjoint group because no intersection and inclusion happen between disjoint prefixes, reducing the overall complexity.

1. Identifying disjoint prefix groups: As a first step, Lupe identifies groups of disjoint destination prefixes in forwarding rules. To this end, Lupe classifies header constraints and their forwarding actions (e.g., forward/drop/rewrite) in the forwarding rules into a radix tree on the basis of their destination prefixes. Lupe then computes subtrees on the radix tree satisfying the following conditions as disjoint destination prefix groups:

1. The destination prefix of a top node in each subtree includes destination prefixes of all the other nodes in the subtree.

2. No destination prefix in a subtree intersects with any destination prefix in other subtrees.

Note that we do not consider the default route in this process as a standard node but as a special top node in all subtrees (represented by *defaultPg* in Algorithm 2) for effective prefix grouping[1]. Since we use the radix tree, we can derive such subtrees from the radix tree using a depth-first search (DFS) from top to bottom. After the derivation, we take all header constraints and forwarding actions in each subtree into a vector, which is the input *pg* of Algorithm 2. Due to the nature of DFS, the header constraints are stored in the vector in order of lower to higher destination prefixes. We leverage this prefix order in the next step to reduce the complexity of computing header spaces.

[1]This rarely happens in real-world; If prefixes with very short prefix lengths exist, such as /1 and /2, we set a parameter k and treat destination prefixes length less than k as well as the default route. We only set the parameter (5) for ACLs in §4.6.1.

2. Computing header spaces by disjoint prefix groups: After identifying the subtrees, Lupe computes a header space for each subtree by analyzing intersection and inclusion relations between header constraints in the subtree. In addition to the prefix grouping, Lupe further reduces the number of the relation checks by adjusting the order of the checks. Lupe retrieves header constraints from a subtree in order of lower to higher destination prefixes and inserts them into a lattice while proceeding with the relation checks in order of higher to lower destination prefixes of nodes in the lattice. This ordering enables Lupe to prune all the subsequent checks if the start of the destination prefix of an inserted header constraint is greater than the end of the destination prefix of the constraint of a currently checked node in the lattice. For instance, if the destination prefix of the inserted constraint is 1.0.0.5/32 and the prefix of the currently checked node in the lattice is 1.0.0.0/30, we can omit all the subsequent relation checks; because the start of the prefix of the inserted constraint (1.0.0.5) is greater than the end of the other's (1.0.0.3) and all the subsequent nodes in the lattice ($\leq$ 1.0.0.3), which indicates no intersection and inclusion relation anymore.

Concretely, we design Algorithm 2 to compute header space based on a lattice for a disjoint prefix group. Each node in the lattice includes a symbolic header constraint and forwarding action (e.g., forward/drop/rewrite) on each network device. Since the header constraint of the default route (or header constraints with destination prefix lengths less than prefix parameter k) could overlap the constraints in all prefix groups, we precompute the header space on a lattice (*defaultLattice*) for the default route and inherit it in computing header spaces for the prefix groups (lines 2-6). To create header space for a prefix group, the algorithm iterates the insertion of each header constraint and forwarding action *pg[i]* (lines 9 to 14). Remember that the destination prefixes in the *PrefixGroup* are in order of lower to higher prefixes. If a header constraint already exists in the lattice, the algorithm only adds forwarding actions in the corresponding node (lines 10 to 12). If not, it (1) creates a new node with the constraint and actions and (2) inserts the node into the lattice (lines 13 to 14). The insertion function (*Insert*) checks if the new node has an inclusion or intersection relation with the existing nodes in the lattice (lines 17 to 31). We extend traditional insertion algorithms on lattice [89,158] for our optimization. The key is to iterate the relation checks for all children of current nodes from *the end* (line 19). Since we insert new nodes into the lattice in order of lower to higher prefixes, we can omit the checks with the subsequent children if the start of the destination prefix of the newly inserted node is greater than the end of the prefix of the child (lines 20 to 21). If not, the algorithm checks the inclusion and intersection relations using Definitions Definition 2 and Definition 3 (lines 22 to 31). In lines 32 to 36, it adjusts parent-child relations in the lattice affected by the new node.

3. Merging subspaces in single header space: Finally, Lupe merges all header spaces of disjoint prefix groups into a single header space by replacing the top (default route) and bottom nodes ($\perp$) on these spaces with the two nodes on the single space. As header spaces of the prefix groups represent all possible headers within their destination prefix ranges, the merged single space symbolically covers all possible headers in given forwarding rules (the proof is in §B.1).

Computational complexity analysis: The computational complexity of this algorithm depends on the number of disjoint prefix groups. For each prefix group, this algorithm needs $\mathcal{O}(N_k^2)$ intersection and inclusion relation checks where N_k is the number of header constraints of the k-th prefix group. In short, the algorithm distributes the relation checks for all the

Algorithm 2: Header space computation based on lattice for a disjoint prefix group

```
   Input: PrefixGroup pg (a vector of header constraints and
          actions)
   Output: a lattice for pg lattice
 1 Function Start(pg):
 2     if defaultLattice = empty then
 3         defaultLattice ← new
 4         defaultLattice.top ← defaultRoute
 5         CreateLattice(defaultPg, defaultLattice)
 6     lattice ← defaultLattice
 7     return CreateLattice(pg, lattice)
 8 Function CreateLattice(pg, lattice):
 9     for all pg[i] in pg do
10         if pg[i].header already exists in lattice then
11             add pg[i].action in the node
12             Continue
13         newNode ← pg[i]
14         Insert(lattice.top, newNode)
15     lattice.bottom ← ⊥
16     return lattice
17 Function Insert(node, newNode):
18     nodeVec ← new // vector of lattice nodes
19     for all child in node.children // loop start at the end do
20         if child.hdr.dstPrefix.end <
             newNode.hdr.dstPrefix.start then
21             break // e.g., 1.0.0.0/30 (.3) < 1.0.0.5/32
                 (.5)
22         if newNode.hdr ⊇ child.hdr then
23             nodeVec.add(child)
24         else if child.hdr ⊇ newNode.hdr then
25             Insert(child, newNode)
26             return
27         else if child.hdr ∩ newNode.hdr ≠ ∅ then
28             create itrNode using intersected hdr. constraint
29             nodeVec.add(itrNode)
30             if the intersected constraint is new in lattice
                 then
31                 Insert(child, itrNode)
32     node.children.add(newNode)
33     checkNode ← {n ∈ nodeVec | ∀m ∈ nodeVec : m ∩ n =
         ∅ ∨ n ⊇ m} // adjust parent-children relation
34     for all cnode in checkNode do
35         node.children.del(cnode)
36         newNode.children.add(cnode)
```

header constraints across each prefix group. In general, this grouping effectively reduces the complexity of these relation checks because operators manage network instances (e.g., routers and applications) using different prefixes for each role, type, and/or physical location. For instance, it divides 2.6×10^4 prefixes into 653 prefix groups in a production datacenter network, which reduces the total number of the relation checks by a factor of 6000x compared to no grouping (§4.6.1). We also confirm its effectiveness in *randomly generated* 5-tuple ACLs (§4.6.1).

4.4.2 Path space computation

Path space is an edge-labeled directed graph to represent the forwarding paths of all headers in header space as illustrated in Figure 4.2. After computing the header space, Lupe first concatenates the forwarding actions on each network device of the parent nodes in the lattice with the actions of their children. As packet forwarding follows the longest-prefix match principle, it only inherits the actions that the children do not have from their parents. In case of packet drop actions (i.e., ACLs), if the parents have the actions on some network devices, their children also inherit them because the range of ACLs in the parents includes the header constraints of the children. Then, Lupe represents the forwarding paths of each lattice's node using the forwarding actions on the directed graph by assigning their unique labels on the header space to edges on the graph.

$$
\begin{aligned}
N &\in \textit{Natural number} \\
\textit{device\_id} &::= \textit{string for a device identifier} \\
\textit{path} &::= \textit{device\_id} \mid . \mid \textit{path}^* \mid \textit{path}\{N\} \mid \textit{path}\{N_1, N_2\} \\
&\quad \mid \textit{path} \mid \textit{path} \mid !\textit{path} \mid \textit{path path} \\
\textit{hdr\_field} &::= N.N.N.N/N \mid [N_1 : N_2] \mid (N_1 : N_2) \mid (N_1, ..., N_n) \\
\textit{hdr} &::= [\textit{hdr\_field}_1, ..., \textit{hdr\_field}_n] \mid !\textit{hdr} \mid \textit{hdr hdr} \\
\textit{query} &::= \textit{hdr} \mid \textit{path} \mid \textit{query and query} \mid \textit{query or query}
\end{aligned}
$$

Header type	Lupe query	
Single field only	[DstAddr:2.0.0.0/24]	
5-tuple with set and range	[SrcAddr:2.0.0.0/24, SrcPort[200-250], DstPort:(88,443)]	
Encapsulated header	[DstAddr:2.0.0.0/24] [DstAddr:3.0.0.0/24]	
Path type	**Lupe query**	**Properties related to query**
Paths reaching r1 to r7	r1 .* r7	Reachability
Paths not reaching r1 to r7	r1 .* !r7	Blackhole, Isolation
Paths reaching r1 to r7 through r5	r1 .* r5 .* r7	Reachability w/ waypoint
Paths reaching r1 to r7 in 2-4hop	r1 .{1,3} r7	Reachability w/ hop constraint
Paths not reaching r1 to r7 in 3hop	r1 .{2} !r7	Hop consistency violation

Figure 4.3: Abstract syntax of the query language in Lupe and query examples.

4.5 Query language and Processing

In this section, we first present the query language of Lupe. We then introduce the algorithms to extract forwarding behaviors matching given queries from header and path spaces.

4.5.1 Query language with example

Operators can query Lupe to extract forwarding behaviors via two query types: header and path queries. Both queries accept symbolic descriptions to support realtime troubleshooting in which operators only have a partial view of current forwarding behaviors. Figure 4.3 presents the abstract syntax and query examples. As described in the examples, the language covers practical network properties including reachability, forwarding loop, blackhole, isolation, waypoint, and hop limit. We demonstrate the expressiveness and usefulness through the two case studies in §4.6.3 and §4.6.4.

Header query: A header query is symbolically defined on header fields. Source and destination addresses are specified by prefixes, and other header fields, such as port and protocol numbers, are specified by ranges or sets. Lupe currently supports 5-tuple, MAC address, and VLAN ID. Operators can define encapsulated headers by describing the inner and outer headers side-by-side (e.g., third row in header queries in Figure 4.3). Unspecified fields in header queries are treated as wildcards.

Path query: A path query is defined by a complete or partial sequence of network device identifiers (e.g., name and router-id) with hop counts of each hop in the paths. The hop counts are quantified by a specific hop (path$\{N\}$), a range of hops (path$\{N_1, N_2\}$), or zero or more hops (path*). This quantification enables operators to specify path queries where hop counts are sensitive (e.g., the consistency of total hop counts of multiple paths in CLOS networks [159]), unlike specification languages in existing verification and synthesis techniques [43,93,104,129].

4.5.2 Header query processing

Given header queries, Lupe extracts all the matching headers from the header space. For extraction, we leverage the lattice's property that is closed under intersection. Because the

header constraints of nodes in the lattice are included by their parents' constraints, the size of the header sets becomes smaller from the top node to the bottom. Leveraging this property, we develop a breadth-first search (BFS) on the lattice from its bottom node to extract the minimal number of headers matching the queries. The algorithm only searches the lattice nodes related to destination prefixes of the given queries (if specified), leveraging the fact that the nodes are arranged by disjoint prefix groups (§4.4.1).

We start the search process for each constraint of parent nodes of the bottom (e.g., nodes ③, ④, ⑤, ⑦, then ⑧ in Figure 4.2). We need two ordered relation checks on every step of exploring the lattice upward. First, we check if the constraint of a currently checked node includes the query's constraint. If so, we record that the node's constraint matches the query and terminate the subsequent process since we already concatenated the forwarding paths of its parents with those of its children in header and path space computation. Second, we check if the constraint of the node intersects with the query's constraint. If intersected, we record that the node's constraint matches the query. We repeat this search process until the search is terminated or reaches the top node in the lattice. We provide the algorithm of this search in §B.2. Note that the header space is independent of the path space, so the search states in the BFS only depend on the total number of nodes in the lattice (i.e., no state explosion happens even in large-scale networks (§4.6.2)).

In terminating the search process, we obtain all symbolic headers matching given header queries. We retrieve forwarding paths of these headers from the path space using the header's labels. These forwarding paths are stored in a hash table with the labels, so we can extract each path in $\mathcal{O}(1)$.

4.5.3 Path query processing

Given path queries, Lupe searches path space to extract forwarding paths matching the queries. For efficiency, we reduce the search space to its subspace that is relevant to the queries. To this end, we treat path queries as search constraints on the path space and terminate the subsequent search when the constraints are no longer satisfied. Since a path query consists of a sequence of device identifiers and hop constraints between the identifiers, we convert them into two constraints: (1) nodes on the path space to be traversed (node constraint) and (2) search depths between the nodes (depth constraint). For instance, the query [r1 .* r7] in Figure 4.2 is translated into the node constraint [r1, . (any device), r7] with the depth constraint [≥ 0]. We record traversed nodes and current search depths in every search step and check if the current state satisfies the two constraints.

We explain the search process with the path query in Figure 4.2. We start the search from (r1) with the node constraint (. and r7) and the depth constraint (≥ 0). Until we can decide if the constraint is satisfied or violated, we do DFS by following labels on the outgoing edges of r1 (labels 1 to 4, 5 to 7, and 8). For instance, the search for label 5 is terminated at (r6) because the third node constraint (r7) is violated at (r6). Conversely, the search for labels 6 and 7 satisfies the node and depth constraints. We then record the path "(r1)→(r3)→(r6)→(r7)" as a satisfied path. We repeat this search process until there are no available paths. Finally, we retrieve header constraints of the satisfied paths from header space using the path's labels to construct forwarding behaviors matching the query.

Backward search optimization: If the number of source nodes is greater than that of

Table 4.2: Dataset description.

Name	Type	#Device/Link	#Rule
INet2	WAN	9/27	77.4k
B4-13	WAN	12/74	77.9k
Stanford	LAN	16/74	3.84k
AT1-2	WAN	16/26	96.0k
B4-18	WAN	33/56	211k
BTNA	WAN	36/76	252k
NTT	WAN	47/73	198k
AT2-2	WAN	68/158	456k
Oteglobe	WAN	93/103	722k
5-tuple ACLs	Backbone routers	N/A	{1,10,100,1000}k
Production DCN	Three-tier Clos	2296/28395	855k

destination nodes in path queries, we reversely traverse the graph while treating the queries in reverse to reduce the search costs. Since the graph search is based on DFS, the search states for a destination node from a source node, such as already traversed nodes and their search depths, can be shared with searches for other destinations from the source. The backward search prunes the redundant searches when answering the queries in large-scale networks with many forwarding paths (§4.6.2).

4.6 Evaluation

We fully implemented Lupe in 6k lines of C++ code and evaluated it on various sizes of synthesized and real network datasets to answer the two important questions:

1. How *scalable* Lupe is for large complex networks? We show that Lupe extracts and outputs forwarding behaviors matching various types of queries from all possible behaviors in 15 seconds, even in a large production datacenter network, 5x-129x improvement over techniques applying existing algorithms for data plane verification (§4.6.1 and §4.6.2).

2. How *useful* is Lupe for troubleshooting real network failures? We conduct two case studies with network operators to demonstrate that Lupe helps them identify the cause and impacts of real failures in a campus network and production datacenter network [160] (§4.6.3 and §4.6.4).

We run all the experiments 10 times on a single core in a computer equipped with an Apple M2 Max processor with 32 GB RAM, and omit their standard deviation because they are small.

Dataset for scalability evaluation: Table 4.2 shows the datasets for evaluating the scalability. To consider troubleshooting scenarios in various types of networks, we use FIBs and network topology of synthesized LANs [161], WANs [161], and real production datacenter networks [17]. We also use four synthesized ACLs to evaluate the efficiency in computing and searching header space with multiple header fields. These ACLs are targeted for real backbone routers in WANs, which are the same datasets in [162] generated by ClassBench [163].

Table 4.3: Header space computation time in LAN/WAN and ACL, and production DCN (s: sec, m: min, and h: hour).

Name	HSA	BDD	Trie	Lattice	Lupe
Stanford	0.01s	0.5s	0.03s	0.02s	0.02s
INet2	1.3s	2.8s	0.4s	2.7s	0.4s
B4-13	1.7s	2.6s	0.4s	0.9s	0.3s
AT1-2	2.4s	2.9s	0.5s	0.8s	0.5s
NTT	6.6s	6.2s	1.1s	1.1s	1.2s
B4-18	7.4s	6.7s	1.2s	1.2s	1.3s
BTNA	8.4s	8.1s	1.3s	2.3s	1.4s
AT2-2	17.1s	15.0s	2.8s	2.0s	2.8s
Oteglobe	37.9s	23.4s	4.0s	23.4s	4.2s
Prod. DCN	>8h	28.1m	1.1m	1.3m	13.3s
ACL1k	N/A	0.8s	0.9s	0.2s	0.01s
ACL10k	N/A	1.3m	8.7s	1.3s	0.1s
ACL100k	N/A	4.2h	24m	5.5m	1.5s
ACL1000k	N/A	>8h	>8h	>8h	1.8m

Method to compare: We replicate the symbolic extraction approach using four frameworks/data structures of existing verification techniques in C/C++: Header Space Analysis (HSA) [58], Atomic predicates based on BDDs [41–43], Multidimensional Trie [45], and (naive) Lattice [47,89].

4.6.1 Header and path Space Computation Time

First, we show that the header and path space computation of Lupe scales for large-scale networks and complex ACLs with multiple header fields. For header space, we measure the time to compute all possible packet headers for each dataset. Table 4.3 gives the results. For small to medium-sized LANs/WANs, Lupe's results are comparable to the other methods replicating our approach. In contrast, only Lupe is scalable for large datasets (ACLs and Production DCN). Unlike the existing verification techniques, which require intersection and inclusion relation checks between header constraints per forwarding path or header constraint, Lupe distributes these checks across disjoint prefix groups. This technique reduces the total number of these checks, which achieves high scalability in the datasets. For instance, it creates 653 prefix groups in 2.6×10^4 prefixes of the production DCN and reduces the number of the relation checks by a factor of 6477 compared to no grouping (i.e., Lattice in Table 4.3). In addition, it creates 998 prefix groups and reduces the checks by a factor of 8.5 in the randomly generated 100k ACLs. We also confirm that the path space computation takes less than a second in all the datasets since it just assigns nodes' labels in header space to edges on the graphs.

These results show that our header and path space computation based on disjoint prefix grouping can support realtime troubleshooting in large-scale networks and with complex ACLs. Note that this technique offers no optimization in L2 networks because MAC addresses have no prefix, but we confirm the efficiency of Lupe in the networks via the case study in a campus network (§4.6.3).

Table 4.4: Processing time for header queries.

(a) Queries based on destination prefix.

Name	BDD	Trie	Lattice	Lupe
Oteglobe	(/24) 6ms	(/24) 0.2ms	(/24) 3.9ms	(/24) 0.2ms
Prod. DCN	(/8) 23.0s (/32) 54.0ms	(/8) 30.5ms (/32) 0.02ms	(/8) 90.6ms (/32) 6.4ms	(/8) 94.1ms (/32) 2.0ms
ACL1k	(/3) 5.6s (/32) 5.5s	(/3) 1.1ms (/32) 0.5ms	(/3) 0.8ms (/32) 0.3ms	(/3) 1.5ms (/32) 0.5ms
ACL10k	(/2) 2.5m (/32) 2.4m	(/2) 23.8ms (/32) 5.5ms	(/2) 10.9ms (/32) 5.6ms	(/2) 11.5ms (/32) 5.7ms
ACL100k	(/2) 12.6m (/32) 7.4m	(/2) 0.7s (/32) 19.8ms	(/2) 0.1s (/32) 19.5ms	(/2) 0.1s (/32) 19.0ms
ACL1000k	N/A	N/A	N/A	(/2) 1.3s (/32) 13.8ms

(b) Queries based on source prefix.

Name	BDD	Trie	Lattice	Lupe
ACL1k	(/2) 10.6s (/32) 10.2s	(/2) 2.7ms (/32) 0.8ms	(/2) 3.3ms (/32) 0.2ms	(/2) 3.1ms (/32) 0.2ms
ACL10k	(/2) 5.7m (/32) 5.7m	(/2) 29.9ms (/32) 5.1ms	(/2) 12.1ms (/32) 0.6ms	(/2) 10.8ms (/32) 0.6ms
ACL100k	(/2) 9.4m (/32) 9.2m	(/2) 0.3ms (/32) 35.8ms	(/2) 0.14ms (/32) 6.2ms	(/2) 0.1s (/32) 5.1ms
ACL1000k	N/A	N/A	N/A	(/2) 1.2ms (/32) 18.6ms

4.6.2 Query Processing Time

Next, we show the efficiency of Lupe in answering header and path queries. In the following experiments, all the outputs (i.e., forwarding behaviors matching queries) go to /dev/null.

Answering header query: We first measure the processing time to search and output forwarding behaviors matching header queries in header spaces. For the baseline performance, we randomly select the shortest and longest source or destination prefixes (other fields are wildcards) except the default route in the datasets as header queries. We do not describe the processing time for datasets where the header space computation timed out (Not Available) [2]. Note that Lattice in this experiment uses the same search algorithm in §4.5.2 because the previous works do not have the algorithm given symbolic headers [47,89]. Thus, its processing time should show a similar trend to Lupe.

Table 4.4a and Table 4.4b show the results for each type of the prefix. The results of Oteglobe show a similar trend to those of other WANs/LANs, so we only show Oteglobe's results. While Lupe creates the header space based on destination prefixes, it answers header queries with specified source and destination prefixes within 1.5 seconds. Because the queries with the shortest prefixes are one of the broadest range queries, this result indicates that Lupe can answer any type of header query in such a timescale.

Answering path query: To measure the processing time to answer path queries, we

[2]HSA is not applicable for this measurement except destination prefix in Oteglobe (Table 4.4a). Its processing time is 18.5ms.

Table 4.5: Processing time for path queries (sec).

Query Type	(1) Src & Dst	(2) Src (edge)	(3) Src (center)	(4) Dst (edge, nop)	(5) Dst (edge, op)	(6) Dst (center, nop)	(7) Dst (center, op)	(8) Mid
Oteglobe	0.03	0.58	0.48	0.39	0.12	1.18	0.37	3.52
Prod. DCN	0.45	0.96	1.01	132.4	0.10	162.4	0.11	661.54

use the largest synthesized WAN (Oteglobe [161]) and production DCN [17] in Table 4.2. For the baseline processing time of various path queries, we pick up eight types of queries in Table 4.5: (1) both source and destination nodes of the path specified (Src & Dst), (2-3) only source node specified where the node is placed on edge (Src (edge)) or center (Src (center)) in the topology, (4-7) only destination node specified without or with optimization in §4.5.3 (Dst (edge, nop), Dst (edge, op), Dst (center, nop), Dst (center, op)), and (8) only intermediate node of the path specified (Mid) (e.g., .* r3 .*).

Table 4.5 shows the time to search and output forwarding behaviors matching the queries. As we can see, Lupe can process almost all queries in a few seconds [3]. In the queries specified only the destination node (4-7), the reverse graph search in §4.5.3 makes the processing time 1000x faster than the search without it in the production datacenter network. This improvement comes from the reduction of search complexity, as there are many source nodes in the large-scale network.

In summary, these results indicate that Lupe can efficiently answer both header and path queries to identify the causes and impacts of network failures in realtime troubleshooting.

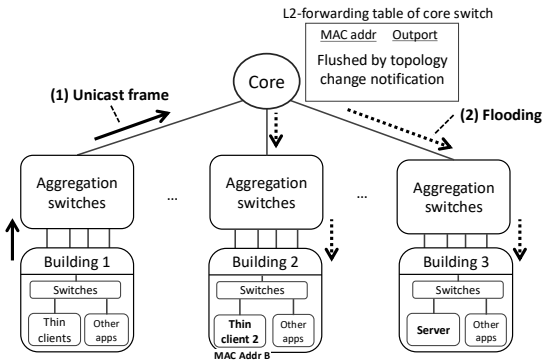
4.6.3 Case study 1: Unknown performance degradation in a campus network

To show the usefulness of Lupe in real troubleshooting, we first demonstrate that Lupe helps operators fix a performance degradation issue in a real campus network. This network is the backbone network of a large national university with over 30,000 students/faculty.

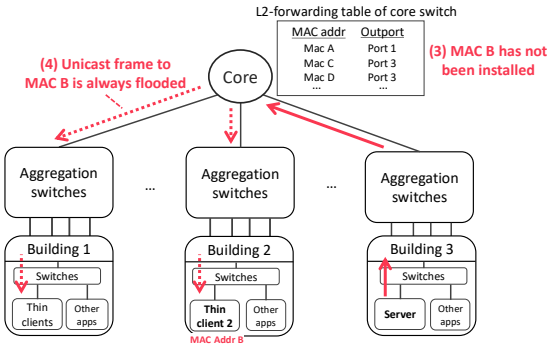
Detail of the failure: One day, a network operator received reports from end users that their thin clients got slow. The operator started investigating connections between the thin clients and their servers. After troubleshooting for a few months, he finally found the cause: a core switch failed to install Layer-2 (L2) forwarding table entries—MAC addresses and output ports—for the affected clients when spanning tree protocol detected topology changes.

Figure 4.4 illustrates how this problem happens. Over several thousand thin clients are accommodated in a single VLAN where spanning tree protocol runs. The core switch, which is the root bridge, receives topology change notifications (TCN) from other switches when the topology changes anywhere. A TCN forces the core switch to flush the forwarding table to relearn MAC addresses and their output ports. When the core switch receives a unicast frame, it updates the forwarding table with the entry of the source MAC address and the ingress port of the frame. The switch next floods the frame to ports other than the ingress port if the destination MAC address does not exist in the table (Figure 4.4a). The TCN flushing

[3]The scope of the eighth query (Mid) in the DCN is very vague and rarely used in practice, but we use it to evaluate the baseline processing time for waypoint queries. Lupe takes most of the time to output the paths matching the query.



(a) When the core switch whose L2 forwarding table is flushed by TCN receives a unicast frame, it floods the frame to ports except the ingress port and installs the source MAC and ingress port of the frame to the forwarding table.



(b) Unicast frames to MAC B are always flooded because the core switch failed to install its MAC address entry into the table due to a massive amount of re-learning for all the addresses after topology changes.

Figure 4.4: Illustration of how the core switch fails to install MAC addresses for clients in the campus network.

the forwarding table causes a massive amount of re-learning for all the MAC addresses in the VLAN; consequently, the core switch sometimes fails to install MAC address entries into the forwarding table (MAC address B in Figure 4.4b) during this process. As a result, unicast frames from the server to the MAC address B are always unknown unicast for the core switch, and the core switch always floods the frames. The flooding makes connections between the server and the thin client with the MAC addresses B slow.

How operators fixed this failure: Since the operator received the reports from end users, he investigated the failure through manual probing and monitoring as follows:

1. He connected his laptop to the VLAN and continually captured packets.
2. He then found unicast frames not supposed to arrive at the switch port to which his laptop was connected, suggesting that the frames were being flooded as unknown unicast.
3. While investigating why flooding occurred, he manually checked all switches (e.g., forwarding table entries and spanning tree protocol).
4. He finally found something wrong happened at the core switch: its forwarding table lacks entries for some existing thin clients (as with the MAC address B in Figure 4.4).

Identifying a switch flooding the unknown unicast frames is time-consuming because many switches could be involved in the network. After receiving the end user's reports, the operator took a few months to identify the cause. Supporting this troubleshooting using data plane verification tools is difficult because they are not designed to extract forwarding behaviors with partial path information related to the failure (the connection between the thin client and its server gets slow).

How Lupe helps operators fix this failure: We confirm the usefulness of Lupe through a troubleshooting simulation with the operator to identify the cause and impacts of

this failure using 3.3×10^5 FIBs (MAC address tables) of 16 core switches in the network. When starting troubleshooting, the operator only knows that the connection between the thin client (Thin client 2) and the server is slow. He gives three path queries to Lupe for identifying the cause and impacts:

1. To begin with, he checks the reachability between the thin client and the server by giving path queries [server .* thin_client_2] and [thin_client_2 .* server] to Lupe.

2. The output of the first query from Lupe includes a forwarding path affected by the lack of a MAC address table entry in the core switch, indicating that packets from Thin client 2 to the server could be flooded at the switch.

3. Next, the operator wants to check if the other thin clients face the same trouble by giving the path query [server .* all_thin_clients] to Lupe.

4. The output of Lupe does not show forwarding paths affected by the lack of the table entry involving the other clients, indicating that the failure only affects Thin client 2 at present.

The operator confirms that Lupe helps (1) reduce the burden of the manual investigation on switches that cause the failure (processes 1 and 2) and (2) identify the failure impacts (processes 3 and 4) with partial path information observable in troubleshooting. We also confirm that Lupe computes the header and path spaces and answers the three queries in less than 1 second.

4.6.4 Case study 2: Silent packet drop in a production datacenter network

As the second case study, we leverage Lupe to help operators detect silent packet drops in a real production datacenter network [160]. We also conducted this study with an operator of the network.

Figure 4.5 illustrates the simplified overview of the packet drops. In this network, 68 NAT servers are deployed in a distributed manner to provide scalable and robust NAT services for endpoint applications. Each NAT server has a designated range of source addresses for performing address translation (rewrite rules in Figure 4.5). If a NAT server receives packets not in its designated range, it redirects them to another NAT server responsible for translating them (redirect rules in Figure 4.5) using IPIP [164]. For robustness, if a NAT server goes down, its address translation ranges are automatically delegated to other NAT servers.

Detail of the failure: While such a distributed mechanism is inevitable for scalability and robustness in large-scale networks, the operators faced silent packet drops (Figure 4.5). The cause was a blackhole that occurred in delegating the address translation ranges. When NAT2 in Figure 4.5 server went down, its address translation range (rewrite App2 to srcB) was delegated to NAT3, and the other NAT servers updated their redirection rules to consider the delegation. However, updating the redirection rule of NAT1 failed due to the high CPU utilization by other processes, which redirects packets to NAT2 that went down. The wrong redirection occasionally happens because this network uses Equal Cost Multi Path (ECMP),

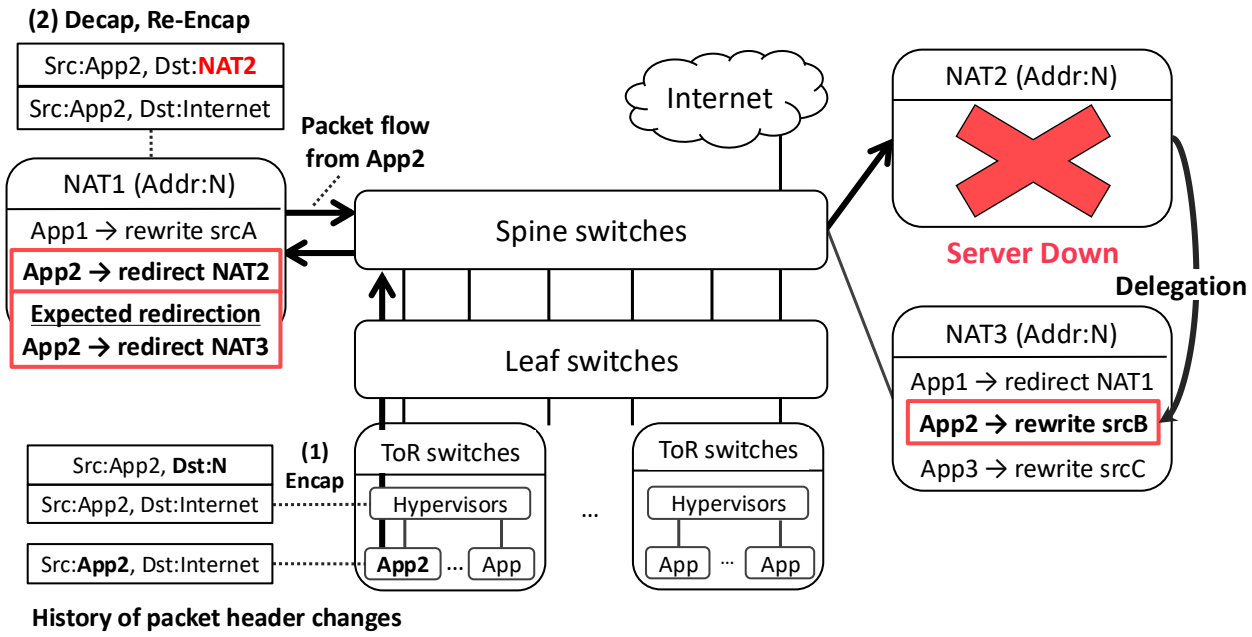


Figure 4.5: Simplified illustration of silent packet drops in a production DCN. For simplicity, NAT X is in charge of the address translation of App X.

and the packets are redirected correctly if forwarded to the other NAT servers that correctly update their redirection rules.

How operators fixed this failure: After the NAT2 server (in Figure 4.5) went down, network operators overlooked the packet drops for a few months since the drops were occasional, and they thought the delegation of the address translation ranges correctly worked. One day, an operator happened to find that packets transmitted from the application (App2 in Figure 4.5) were occasionally dropped in his daily operational work. The workflow to investigate the cause was as follows:

1. The operator investigated how packets transmitted from App2 were forwarded.
2. He found that the packets were forwarded to NAT2, which was already down.
3. To identify the causes, he manually investigated NAT/IPIP rules in other NAT servers.
4. Through the investigation, he found that a redirection rule in NAT1 failed to be updated.

Manually identifying the causes of the packet drops and the other impacts is hard because their NAT servers have 1.16×10^6 NAT/IPIP rules with various match conditions. After noticing the packet drops, the operator took a few hours to identify the NAT server. Supporting this troubleshooting process using the verification tools is also difficult because operators only know that the NAT2 server went down and do not know what forwarding properties should be satisfied by the verification.

How Lupe helps operators fix this failure: We leverage Lupe with an operator of the network to identify the impacts of NAT2 server down using the 1.16×10^6 NAT/IPIP rules and FIBs in NAT servers and routers. The troubleshooting processes with Lupe are as follows:

1. To identify the impacts of NAT2 server down, the operator first checks forwarding behaviors involving NAT2 using the path query [All_Applications .* NAT2].
2. The output of Lupe shows forwarding paths indicating that packets transmitted from App2 are wrongly redirected to NAT2 by a redirection rule of NAT1.
3. Next, he checks how packets from App2 are forwarded through other NAT servers by giving the header query [SrcAddr:App2] to Lupe.
4. The output of Lupe indicates (1) NAT1 redirects the packets to NAT2 and (2) the other NATs redirect them to NAT3, which confirms that the other NATs correctly redirect the packets.

The operator confirms that Lupe helps (1) notice the packet drops induced by NAT2 server down (processes 1 and 2) and (2) identify the impacts of the server down (processes 3 and 4) with the partial headers and paths observed in the troubleshooting. Lupe takes 74.7 seconds to compute the header and path spaces and less than 1 second to answer the two queries, using 3 GB of memory for the computation. Thus, we confirm Lupe's scalability in real troubleshooting for large-scale DCNs.

4.7 Limitation and Discussion

Limitation of Lupe. Lupe has two limitations in identifying the causes and impacts of network failures. First, Lupe can not consider *dynamic* forwarding properties such as latency and packet drop rates since it statically computes possible behaviors using forwarding rules and network topology (as discussed in Table 4.1). Net2Text [145] and Anima [117] answer queries about such dynamic behaviors by collecting traffic states on each forwarding path. Second, Lupe can not identify software and hardware bugs that can not be inferred from the forwarding rules (e.g., ASIC and driver bugs). Fault localization techniques allow operators to automatically infer the causes through active probing [62,66,69,148,149,165,166].

Lupe is a complementary approach to these techniques. While Lupe can not support such dynamic behaviors and software/hardware bugs, it enables operators to extract forwarding behaviors matching with partial headers and paths from all possible behaviors that these techniques can not cover. This feature is crucial to troubleshoot network failures caused by faulty forwarding rules due to misconfiguration of routing protocols and ACLs (one of the major causes of network failures [8,9,33,143,153]).

Comparison with existing symbolic analysis tools. Batfish supports symbolic reachability analysis by generating data plane states through control plane simulation using BDDs [32,33]. However, this approach incurs computational overhead due to the simulation of routing convergence. In contrast, Lupe is specifically designed to compute and extract forwarding behaviors from data plane forwarding rules (e.g., FIBs and ACLs) using our proposed data structures and algorithms, such as the lattice-based header and path space computation with disjoint prefix grouping (§4.4) and search space pruning (§4.5). These algorithmic contributions enable Lupe to efficiently (1) compute all possible packet forwarding behaviors and (2) extract forwarding behaviors matching partial headers and/or forwarding paths, ensuring scalability for large-scale networks and complex ACLs (§4.6).

Pre-computing header space. Existing data plane verification pre-computes all possible headers and incrementally updates them by collecting every data plane change (e.g., FIB changes) for sub-millisecond level verification. However, operators face the difficulty of collecting every change due to operational reasons or hardware/software failures [8,93,160,167]. If we miss some changes, modeled forwarding behaviors are inconsistent with real-world behaviors. As Lupe is a complementary tool to the verification, we avoid this problem by computing header space *from scratch* when starting troubleshooting. Since the header space computation of Lupe is scalable to meet the realtime constraint in troubleshooting (§4.6.1), this approach does not burden troubleshooting processes.

Data plane consistency. Data plane consistency is crucial to detect transient failures in troubleshooting [26,48,93,168]. One potential extension of Lupe to this problem is to capture consistent snapshots of data plane [26,48] or leverage a transient control plane simulator [169].

Summarization of forwarding behaviors. Summarizing forwarding behaviors matching given queries can be useful when there are many matching behaviors. Applying existing summarization algorithms (e.g., ComPass [145]) to forwarding paths extracted by Lupe could reduce the cognitive burden on operators by presenting a high-level overview of numerous forwarding paths.

Non-deterministic forwarding. We support non-deterministic packet forwarding (e.g., ECMP) by enumerating possible forwarding paths in creating a path space. While such enumeration may produce many paths in large networks, we show that the enumeration works even in production DCNs (§4.6).

Header transformations. We support header transformations (header rewrite/encapsulation/decapsulation) by concatenating the forwarding behavior of packet headers before and after transformations. However, we currently have no theoretical support for terminating infinite loops with encapsulated headers. We will apply the algorithm finding infinite loops in Ref.[44] in computing forwarding paths.

Stateful network functions. We currently do not support stateful network functions such as stateful firewalls and load balancers. To model forwarding behaviors with such stateful functions in Lupe, network semantic models with stateful functions could be beneficial [87,88,170].

4.8 Summary

This research proposed Lupe, which helps operators identify the causes and impacts of network failures by extracting forwarding behaviors matching given symbolic headers or forwarding paths. We showed that Lupe extracts forwarding behaviors matching various types of symbolic headers or paths in 15 seconds in a large-scale production datacenter network, 5x-129x improvement over techniques applying existing algorithms of data plane verification. In addition, two case studies in campus and production DCNs confirmed the usefulness of Lupe in helping operators identify the causes and impacts of real failures.

Chapter 5

Real-time Data Plane Verification for Large-scale Networks with Header Transformations

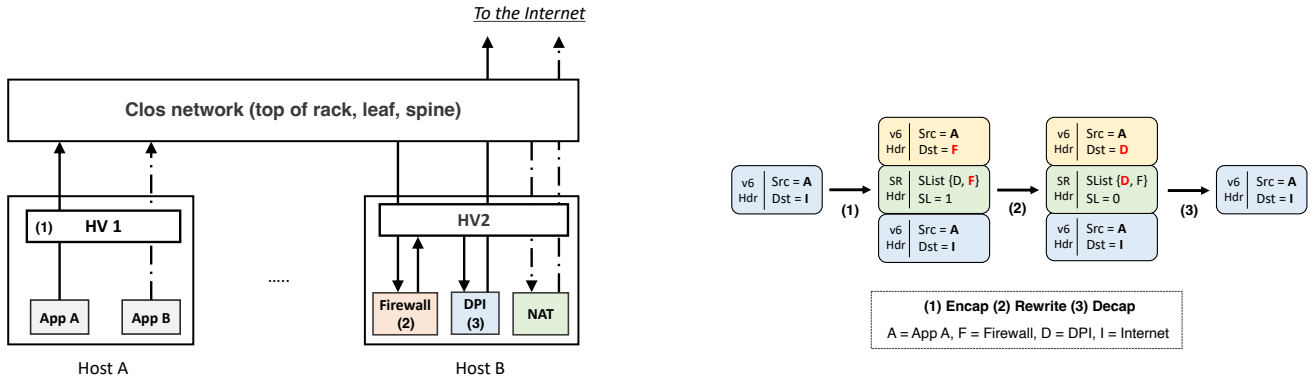
In this chapter, we propose a real-time data plane verification framework that verifies sophisticated network techniques in today's large-scale networks.

5.1 Introduction

Unlike traditional shortest-path-based networking, packet forwarding behavior in today's large-scale networks, such as datacenter networks (DCNs), is tremendously complex. These networks house various applications on their single physical infrastructure, each with different forwarding requirements. Some requirements are performance-related (e.g., high throughput and low latency), while others are functionality-related (e.g., requiring NAT [171,172] and load-balancer [173,174]). To simultaneously meet them, network operators use sophisticated network techniques such as Traffic Engineering (TE) and Service Function Chaining (SFC) [12–16,18,175]. These techniques dynamically customize forwarding behavior for each requirement beyond the shortest paths. Despite the benefits, the operators are required to manage each customized behavior, which is difficult due to the network scale. In addition, the network functions have been deployed in a distributed manner for their scalable and robust provision. Such distributed deployment has caused new operational difficulties (§5.7.4), which leads to unintended network failures.

Data plane verification has been extensively studied to ensure the safety of the complex forwarding behavior [26,41–48,50,58,89,176–178]. It models networks with forwarding rules and automatically verifies network properties on the model, such as reachability and consistency (i.e., no routing loop or blackhole). Operators use the techniques to check if forwarding behavior based on current data plane snapshots satisfies their policies in realtime.

Despite the recent progress, the existing techniques lack the two inseparable features in verifying large-scale networks with sophisticated network techniques: *scalability in realtime verification and expressiveness of modeling the techniques*. These networks house thousands of network devices and hundreds of thousands of forwarding rules [4,8,26]. Furthermore,



(a) (b)

Figure 5.1: A simplified example of SFCs with SRv6 in a production datacenter network [175]. (a) Forwarding behavior of packets from Apps A and B. (b) Packet header transformation from App A. Note that F is Firewall, D the DPI, and I a destination address in (b).

they produce thousands of data plane updates within a second [48]. Thus, we need sub-millisecond level verification to handle each update for this network scale. In addition, the sophisticated techniques deployed on large networks should be precisely modeled. They dynamically change the forwarding behavior by transforming packet headers (i.e., packet rewrite/encapsulation/decapsulation). Modeling and verifying the forwarding with the transformations is complicated because we need to carefully distinguish the forwarding behavior of each transformed header. Achieving the two features is challenging because the scalability largely depends on the model expressiveness [84]. Therefore, the existing techniques only satisfy one of the features (the details in §5.3).

To overcome the challenge, we propose Graft, a realtime data plane verification framework for verifying packet forwarding behavior with the header transformations on large-scale networks [1]. The key design choice is to restrict the scope of header fields essential for forwarding (5-tuple and protocol-specific fields for the transformation) in verification. While the restriction limits network properties to be verified (§5.4), it enables us to develop an optimized data plane model and verification algorithms by combining data structures suitable for the selected fields. For scalable realtime verification, we propose an algorithm leveraging the packet flow characteristics to efficiently compute and manage large header spaces and their forwarding paths (§5.4.1). As the packet flows and their forwarding behavior are based on the longest-prefix match, leveraging this principle naturally enables us to avoid significant overhead in existing verification techniques (§5.6 and §5.7). We combine the radix tree, a suitable data structure to effectively store and search the large header spaces using the principle, and the hash tables for managing their forwarding paths in the algorithm. The header space and path management provide enough scalability to quickly verify production DCNs (§5.7.2). We second propose a graph-based data plane model and algorithms to precisely model the forwarding behavior with header transformations (§5.4.2 and §5.4.3). On the graph model, we model the transformations as transitions between header spaces and

[1]The target size of the large-scale networks is <2,250 network devices and instances (e.g., virtual machines and hypervisors) in our real DCN dataset (§5.7).

encode them into a hash table. These processes enable us to use polynomial-time graph search algorithms to verify forwarding behavior with the transformations, which makes realtime verification for large-scale networks possible (§5.4.4). We theoretically prove the correctness of the search algorithms with formal network semantics.

To show scalability for large-scale networks and expressiveness for the header transformations, we evaluate Graft with synthetic DCNs (§5.6) and a real production DCN (§5.7). Through the evaluation, we first show that Graft verifies network properties 100x faster than the existing approach in the synthetic DCNs and 20000x faster in the production DCN. Second, we demonstrate that Graft has enough expressiveness for the transformations by modeling and verifying SRv6-based SFCs in the DCN. Furthermore, we show that Graft can detect actual network failures caused by the distributed NATs in the DCN. To the best of our knowledge, we are the first to verify the correctness of sophisticated network techniques and distributed network services in production DCNs.

The contributions of this paper are as follows. (1) We propose algorithms and a data plane model with formal semantics to verify packet forwarding behavior with sophisticated network techniques in large-scale networks (§5.4). (2) Through evaluations with synthetic and production DCN datasets, we show that our proposed method achieves both scalability for large-scale networks and expressiveness for header transformations (§5.6, §5.7). (3) We are the first to demonstrate how to verify and debug the sophisticated techniques in the production DCNs (§5.7).

5.2 Motivating Example: SRv6 in DCN

We introduce a simple example of sophisticated network techniques to realize various application demands on large-scale networks. Figure 5.1 illustrates two simplified SFCs that are deployed in a real production DCN [175] [2]. In the network, App A forwards packets to the Internet through two network functions: Firewall and Deep Packet Inspection (DPI). App B forwards packets to the Internet through NAT. To satisfy the different requirements, the network operators use Segment Routing over IPv6 (SRv6) [179]. To forward a packet in accordance with application demands, SRv6 embeds a forwarding path into a segment list field (SList) as an ordered list of IPv6 addresses and a current index for the segment list into a segment left field (SL). As an example, we illustrate packet forwarding behavior from App A to the Internet and the transitions of its header in Figure 5.1. A packet transmitted from App A is first encapsulated with an IPv6 header and an SR header on HV1 ((1) in Fig Figure 5.1a). When the packet reaches Firewall, Firewall checks the packet for security and rewrites the destination address on the basis of the values of the SList and the SL ((2) in Fig Figure 5.1a). Since the destination address is DPI, the packet is forwarded to DPI. DPI then inspects the content of the packet. Suppose the packet is benign, and DPI decapsulates the outer header and transmits the packet to the Internet because of no other header for the encapsulation in the SR header ((3) in Fig Figure 5.1a). Like this example, operators simultaneously realize multiple forwarding policies on single networks by customizing the forwarding behavior beyond traditional shortest-path routing.

[2] For simplicity, we assume that network devices in the SFCs are SRv6-aware, but Graft also works on SRv6-unaware devices.

Despite its benefits, managing such forwarding behavior is a complex task due to the current network scale and complexity [4,9,22]. Suppose an operator misconfigures the packet processing at HV2, and packets from App A to the Internet are dropped. The operator first notices that App A can not reach the Internet. To pinpoint its cause, the operator uses a testing/monitoring tool to discover that the packets are being dropped at HV2. Afterward, the operator must check whether the forwarding, access control, and SRv6 rules in HV2 are correctly configured. However, such a check is hard because of the huge number of configurations and forwarding rules.

5.3 Challenges

Data plane verification is a promising approach to detecting network failures and their causes in realtime [26,41–48,50,58,89,176–178]. To verify packet forwarding behavior with sophisticated network techniques in large-scale networks, two challenges must be simultaneously overcome: *expressiveness of header transformations* and *scalability for large-scale networks*. We discuss them and why existing techniques can not achieve them.

Expressiveness. Modeling header transformations is essential to verify forwarding behavior with sophisticated network techniques. The fundamental difficulty in modeling them is to accurately treat forwarding behavior with transformed headers on data plane models. With sophisticated techniques, packet headers are dynamically encapsulated/decapsulated and rewritten during forwarding. Thus, data plane verifiers must handle a stack of headers and rewritten headers in their verification. Analyzing forwarding behavior with these headers is complicated because forwarding properties at a certain point depend on the history of how headers are transformed and forwarded before the point (forwarding history). For instance, when we check whether encapsulated headers induce a forwarding loop, we need to clearly distinguish the forwarding history of each header. Consider a packet encapsulated with headers A, B, and A in order. Suppose the packet goes through a router with header A for the second time. Still, different parts of header A in the encapsulated header are used at the first and second traversal, and the packet finally reaches a destination. In that case, the forwarding behavior should not be considered a forwarding loop. If the verifiers do not correctly consider the forwarding history, they produce false negative results in their verification. Therefore, traditional verification algorithms that do not consider header transformations can not correctly model header transformations on their models [26,45–48,58,176]. We need a data plane model and algorithms with theoretical correctness to handle the transformations.

Scalability. Data plane models and their verification techniques should be scalable for verifying properties in large-scale networks in realtime. These networks house a huge amount of network devices and forwarding rules. Moreover, they produce thousands of data plane updates in a second when some data plane condition changes [48]. Therefore, sub-millisecond level verification is inevitable for this scale. Unfortunately, none of the existing techniques have overcome this challenge while meeting the expressiveness requirement.

APT [42] and Katra [44] are the only verifiers that handle three header transformations for sophisticated network techniques [3]. These approaches compute header constraints (header

[3] P-Rex [90] and AalWiNes [91] specialize header transformations of MPLS, but Graft generalizes them to support other protocols/network functions. MNV [178] is a recent work to handle the transformations

spaces) on forwarding paths considering the transformation and develop an algorithm to verify their models with the transformations. For instance, Katra classifies these constraints as some classes called Partial Equivalence Classes (PECs) as follows:

Definition 4. (Partial Equivalence Class) A set of packet header sets $\{H_1, H_2, ..., H_n\}$ are partial equivalence classes if the following holds:

1. $H = H_1 \cup H_2 \cup ... \cup H_n$,
2. $\forall i, j \in \{1, ..., n\},\ i \neq j \implies H_i \cap H_j = \emptyset$,
3. $\forall i \in \{1, ..., n\}, \forall h_1, h_2 \in H_i, \forall r \in R, OutPort(r, h_1) = OutPort(r, h_2) \wedge Trans(r, h_1) = Trans(r, h_2)$,

where H is the set of all packet headers, R is a set of network devices, $OutPort(r, h_i)$ is ports to which h_i is forwarded on r, and $Trans(r, h_i)$ is a transformer that matches h_i on r.

Katra computes a PEC-based data plane model with stacks of packet headers to model its encapsulation and decapsulation. While these approaches handle the transformations, they lack scalability for large-scale networks such as DCNs. The main reason is that they rely on Binary Decision Diagrams (BDDs) [92] to compute and manage the header spaces of Equivalence Classes (ECs). BDD treats packet headers just as sequences of bits and symbolically stores them. In this process, it compresses its internal data representation to efficiently represent the header spaces of each EC (space-time tradeoff). BDD needs this operation every time a forwarding state in the model is updated. As reported in past literature [46,89], the operation in the verification requires significant overhead and does not scale well in DCNs. For instance, BDDs take over 20 minutes to compute the ECs in a production DCN that houses 2200 network devices and several seconds for model updates due to the operation (§5.7). Therefore, these approaches are insufficient to handle the burst data plane updates in large-scale networks.

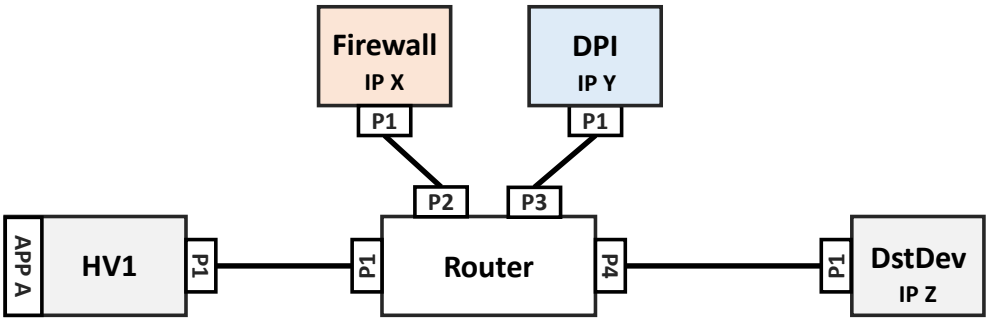
5.4 Our approach

To overcome the two challenges, we propose Graft, a realtime data plane verification framework that simultaneously achieves the expressiveness of header transformations and scalability for large-scale networks.

Expressiveness. To correctly model and verify forwarding behavior with the transformations, we formalize our data plane model and verification algorithms with formal semantics (§5.4.2). Using our semantics, we prove the algorithms' correctness with transformed headers (§C.2 to §C.4). On the model, we handle transformed headers by tracking the forwarding history of each header (§5.4.3). Unlike APT and Katra, we propose a new technique to handle the transformations, which enables us to use polynomial-time graph search algorithms in the verification (§5.4.3).

Scalability. For scalable verification, we make two key observations. First, we can avoid the BDD's overhead in managing header spaces of ECs by leveraging data structures suitable

on modular data plane model; however, it only focuses on the reachability property. Graft supports other network properties essential for the safety of large-scale networks with sophisticated network techniques such as forwarding loops and multipath consistency.



Rule	Dev	InHdr	Type	OutHdr	OutPort
r1	HV1	H	Encap	F	-
r2	HV1	F	Forward	F	P1
r3	Router	F	Forward	F	P2
r4	Router	D	Forward	D	P3
r5	Router	H	Forward	H	P4
r6	Firewall	F	Rewrite	D	-
r7	Firewall	D	Forward	D	P1
r8	DPI	D	Decap	H	-
r9	DPI	H	Forward	H	P1

Figure 5.2: Simple SFC network. After headers are transformed by the rules, the packets are forwarded on the basis of the transformed headers (OutHdr).

for header structures. Since BDD first treats packet headers just as sequences of bits and compresses its data representation for memory efficiency, we can omit the compression process by developing header space computation leveraging the structures. In modeling forwarding behavior, the destination address is the essential field because packet flows are followed by the longest-prefix match principle. Thus, we leverage data structures following the principle in EC computation for efficiently managing a large header space (§5.4.1). We show that our header space management is more scalable and takes similar memory consumption compared with a BDD-based approach (§5.6, §5.7). Second, the required header fields are limited in the network/transport layers. For instance, policy violations of network-level properties such as reachability and isolation are mostly induced by 5-tuple-based packet forwarding [26,172]. Thus, we restrict header fields for verification to 5-tuple and protocol-specific fields for header transformations such as SR header. We then develop algorithms optimized for the fields to verify the large-scale networks.

The processing flow, from model building to verification, is as follows. We first compute ECs with a radix tree and hash tables, which bring scalable header space and path management of ECs for large-scale networks (§5.4.1). To model packet forwarding behavior, we create a graph-based data plane model based on the ECs with formal semantics (§5.4.2). On the model, we treat header transformations as transitions between ECs with hash table and stack (§5.4.3). We verify the operator's policies by developing graph search algorithms on the model (§5.4.4) and proving the soundness of the reachability algorithm (§C.3). To illustrate our approach with end-to-end packet forwarding behavior, we use another simple network shown in Figure 5.2.

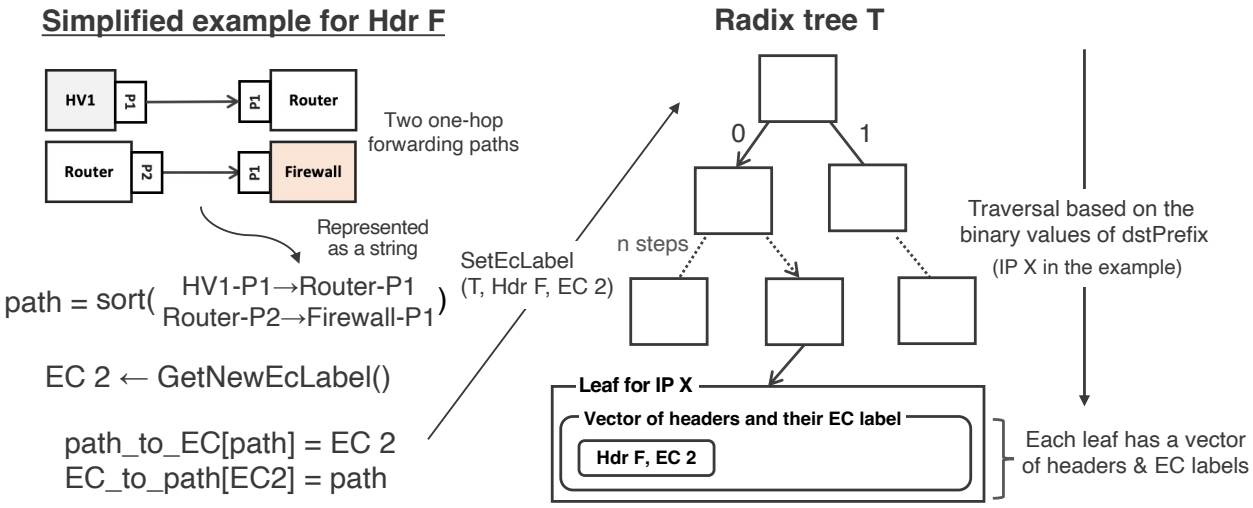


Figure 5.3: Overview of Algorithm 3 in Figure 5.2 network.

5.4.1 Computing path-based ECs

To model packet forwarding behavior, we first compute ECs based on forwarding paths. We define our ECs as follows:

Definition 5. (Equivalence Class) A set of packet header sets $\{H_1, H_2, ..., H_n\}$ are equivalence classes if the following holds:

1. $H = H_1 \cup H_2 \cup ... \cup H_n$,
2. $\forall i, j \in \{1, ..., n\},\ i \neq j \implies H_i \cap H_j = \emptyset$,
3. $\forall i \in \{1, ..., n\},\ \forall h_1, h_2 \in H_i,\ \forall r \in R,\ OutPort(r, h_1) = OutPort(r, h_2)$,

where H is the set of all headers, R is a set of network devices, $OutPort(r, h_i)$ are output ports.

To distinguish an EC from another EC, we assign a unique label called "EC label" to each EC. A major difference between the definition of ECs and PECs in Katra is that our EC definition does not consider header transformations. We will describe how to handle the transformations in §5.4.3.

To avoid the significant overhead of BDD, we propose EC computation and management suitable for header structures. Since the packet flows are followed by the longest-prefix match forwarding principle, we leverage data structures following the principle. In this process, we focus on selected header fields essential for packet forwarding [4]. For header space management, we use a radix tree, a suitable data structure to store and search the headers used for packet forwarding. The EC computation with the radix tree can leverage the locality of IP prefixes created by hierarchical route summarization [10] in production DCNs (§5.7.2). Besides, we use hash tables to manage forwarding paths for each EC. We represent the forwarding paths

[4]We focus on 5-tuple and SR header as selected header fields in this paper. Still, Graft can support other header fields such as Ethernet and VXLAN.

of ECs as a string of network devices and their ports. Therefore, a hash table is suitable for quickly retrieving what paths each EC has and, conversely, what paths belong to which EC.

Concretely, we design Algorithm 3 to compute ECs using a radix tree and hash tables. We illustrate a computing process in Figure 5.3. In the algorithm, we have five assumptions. First, forwarding prefixes in P are collected from forwarding rules in network devices D. Second, a header transformation rule *htr* in *Htr* is a tuple consisting of three components: a network device that has the rule, a key for the rule, and a value for the rule. While we do not consider header transformations in the algorithm, we compute the forwarding paths of headers before and after their transformation. Third, access control rules in A are in the form of a sequence of rules of permitted headers, followed by the rule that denies the others. The algorithm can also handle the converse form of the rules (denying some headers and permitting others) by reversing the processes in lines (L) 15 to 17, 20, and 21. Fourth, each field of packet headers in Algorithm 3 is represented as the symbolic value considering its forwarding characteristics, such as representing destination IP addresses by IP prefixes to compactly represent header spaces. Lastly, a key for the radix tree T is a destination prefix. Each leaf in the radix tree has a vector of headers and their EC labels for which the prefix matches the leaf.

The algorithm first generates a set of headers H to compute their forwarding paths (L 2 to 5). If some header fields are empty in the generation, we treat them as wildcards. We currently do not support policy-based routing, but we can do that by adding their rules, like the process in L 2 and 3.

For each header *hdr* in H, the algorithm computes the forwarding path by checking how the header is forwarded on every network device (L 6 to 19). For each device d, the algorithm adds a one-hop forwarding path of the header to the string *tmp_path* (L 8 to 14). To consider access control rules, it checks if there is an access control rule that denies the header on the device (L 15). If so, it first gets packet headers permitted by the rules (L 16). It then temporarily stores the permitted headers and their one-hop forwarding paths to a vector *Vec_Hdr_Path* (L 17). If the function finds a header overlapping the permitted headers in the traversal, it creates disjoint headers by splitting overlapping and non-overlapping parts of these headers. The split is easily done by applying the binary operation to each field of their headers. If no access control rule denies the headers, the algorithm adds *tmp_path* to the string *path* (L 18 and 19).

After checking all the one-hop forwarding paths, we add the header *hdr* and an empty forwarding path (null) to the vector *Vec_Hdr_Path* to consider a set of packet headers rejected by all access control rules or a situation where there are no access control rules in the network (L 20).

For each header *vhdr* in *Vec_Hdr_Path*, the algorithm checks whether an EC label is already assigned to the path using a hash table *path_to_EC[path]* (L 23). If so, the algorithm gets the EC label from the hash table and stores the label into a leaf of the radix tree T (L 24 and 25). Otherwise, the algorithm first creates a new EC label and sets the label and the path to *path_to_EC[path]* and *EC_to_path[ec_label]* (L 27 to 30). It then stores the label into a leaf of T (L 31).

Finally, the algorithm returns the radix tree T (L 32). We use the resulting radix tree and the hash tables to manage the header spaces and their forwarding paths of ECs. Note that our path-based EC computation returns the minimal number of ECs and prevents EC

Algorithm 3: Computing path-based ECs

Input: a set of forwarding prefixes P, hdr. trans. rules Htr, access control rules A, and network devices D
Output: the radix tree T of which each leaf has a vector of headers and their EC labels

```
1  Hdr, T ← New
2  for p in P do
3      Hdr.add(p)
4  for htr in Htr do
5      Hdr.add(htr.key)
6      Hdr.add(htr.value)
7  for hdr in Hdr do
8      path, Vec_Hdr_Path ← New
9      for d in D do
10         tmp_path ← New
11         output_ports ← GetOutputPorts(d, hdr)
12         for port in output_ports do
13             src ← d.name + port
14             dst ← L2_adjacency[src]
15             tmp_path ← tmp_path + src + dst
16         if Is_Dropped_Hdr(A, d, hdr) ≠ null then
17             ph ← Get_Permitted_Hdr(A, d, hdr)
18             Vec_Hdr_Path.add(ph, tmp_path)
19         else
20             path ← path + tmp_path
21     Vec_Hdr_Path.add(hdr, null)
22     for vhdr in Vec_Hdr_Path do
23         path ← sort(path + vhdr.path)
24         if path_to_EC[path] ≠ null then
25             ec_label ← path_to_EC[path]
26             SetEcLabelToRadix(ec_label, T, vhdr)
27         else
28             ec_label ← GetNewEcLabel()
29             path_to_EC[path] ← ec_label
30             EC_to_path[ec_label] ← path
31             SetEcLabelToRadix(ec_label, T, vhdr)
32 return T
```

explosion produced by prefix-based EC computation like Veriflow [45]. In §C.1, we prove that Algorithm 3 computes ECs satisfying Definition 5.

5.4.2 EC-labelled graph

To model packet forwarding behavior in a network, we represent its data plane as a single edge-labelled directed graph with ECs. To do so, we embed the forwarding paths and their EC labels computed by Algorithm 3 as the directed edges in the graph. We call the edge-labelled directed graph the "EC-labelled graph". We define the EC-labelled graph as follows:

Definition 6. (EC-labelled graph) An EC-labelled graph is a tuple $G = (V, E, ECL, L)$ where:

- V is a set of *vertices*,
- $E \subseteq V \times V$ is a set of *directed edges*,
- ECL is a set of *EC labels*, and
- $L : E \to ecls$ is a function from an edge to a finite set of EC labels on it.

We see V as a set of pairs of network devices and their ports. E is a set of directed edges in the forwarding paths of each EC. We create a function L as follows. We enumerate all forwarding paths and their EC labels by accessing the hash table *path_to_EC[path]*. We then map the directed edges in the forwarding paths to their EC labels in the function L. As an example, Figure 5.4 illustrates the EC-labelled graph modeling the network in Figure 5.2.

Preliminaries for Semantics. We define the semantics of an EC-labelled graph to formally verify network properties on the graph. To this end, we first describe *located labels*. Given a vertex $v \in V$ and an EC label $ecl \in ECL$, we call the pair of v and ecl written by $\langle v, ecl \rangle$ a located label $ll \in LL$ where LL is a set of located labels in an EC-labelled graph. Traversing an EC-labelled graph produces a sequence of located labels to describe the located label history on the graph. Second, we use a unique label called transition label *trl* to identify which headers in an EC are transformed. Third, we describe an EC label transformer *Hs*, which is a function to model header transformations on EC-labelled graphs. It takes a transition label, an EC label, and a vertex as input and returns an EC label. We use the label and the transformer to model the transformations on the graphs in §5.4.3.

Semantics. We define the semantics of an EC-labelled graph G as a function $[\![G]\!]_i : LL \to LL^*$ that takes an initial located label to a trace of located labels through the EC-labelled graph for a given number of steps $i \subseteq \mathbb{N}$:

$$[\![G]\!]_i\langle v, ecl\rangle = \begin{cases} \varepsilon \cdot \langle v, ecl\rangle & \text{if } i = 0 \\ \sigma \cdot \langle w, ecl'\rangle & \text{if } Hs(trl, ecl', w) \text{ is undefined} \\ \sigma \cdot \langle w, Hs(trl, ecl', w)\rangle & \text{otherwise} \end{cases}$$

where $\sigma = [\![G]\!]_{i-1}\langle v, ecl\rangle$, $\text{top}(\sigma) = \langle v', ecl'\rangle$, v' is a vertex connected to w, trl is a transition label, and Hs is the EC label transformer. Here, we define the termination of the graph traversal as follows:

Definition 7. (Termination of graph traversal) An EC-labelled graph G has terminated a located label ll after i steps, written $G \otimes \langle i, ll\rangle$, if the trace no longer changes: $[\![G]\!]_i ll = [\![G]\!]_{i-1} ll$.

Property. We define properties that can be verified on EC-labelled graphs using traces of our network semantics. Since the traces are sequences of vertices and EC labels, we can define and verify arbitrary forwarding properties determined on the sequences. For instance, we define a forwarding loop and a blackhole on the trace as follows. Note that we refer to the loop definition of Katra.

Definition 8. (Forwarding Loop) Given an EC-labelled graph G, an input located label ll induces a forwarding loop if there exists a step-index $i \in \mathbb{N}$ for the start of the loop such that for all steps $j \in \mathbb{N}$ where $j \geq i$, there exists a future $k \in \mathbb{N}$ such that

1. $|[\![G]\!]_j ll| < |[\![G]\!]_k ll|$,
2. $top(hops([\![G]\!]_j ll)) = top(hops([\![G]\!]_k ll))$.

where *hops* is a function that takes the trace of located labels and returns its vertices.

Definition 9. (Blackhole) Given an EC-labelled graph G and a vertex *dst*, an input located label ll induces a blackhole if there exists a trace when G has terminated ll after i steps such that $top(hops([\![G]\!]_i ll)) \neq dst$.

Similarly, we can define and verify other properties obtained from the Graft's trace. Graft currently supports other properties such as hop limit, waypoint, and isolation. We will describe the verification algorithms in §5.4.4.

5.4.3 Modeling header transformations

We model header transformations in an EC-labelled graph to verify packet forwarding behavior with the transformations. We first define the header transformations as follows:

Definition 10. (Header transformation) A header transformation $HT : H \rightharpoonup H$ is a partial function over packet headers.

Intuitively, if packet headers match a rule of the header transformation, these headers are transformed into other headers by the partial function HT. Otherwise, the headers are not transformed because the ranges of HT are not defined.

Unlike APT and Katra, our key idea is to treat header transformations as transitions of EC labels on the graph for fast processing. Concretely, we (1) trace the transformations in packet forwarding using a graph search algorithm and model the traces as transitions of EC labels, and (2) encode them into a hash function. These processes enable us to make the cost of the header transformations during the verification negligible (§5.6.4). We describe the key ideas step by step.

Tracing header transformations

We trace and model the header transformations in packet forwarding as the transition relations of the EC labels on the labelled graph. We define the transition relations as follows:

Definition 11. (Transition relation of EC labels) A transition relation of EC labels is a tuple $TE = (V, InECL, ht, OutECL)$:

- V is a set of *vertices* in EC-labelled graph G,
- $InECL \in ECL$ is an EC label of a header,
- ht is a type of header transformation, and
- $OutECL \in ECL$ is an EC label of a header.

$InECL$ and $OutECL$ are EC labels of headers before and after applying a header transformation ht.

Before explaining the tracing algorithm, we describe how to detect forwarding loops (Definition 8) on EC-labelled graphs to confirm its termination. In the tracing algorithm, we traverse vertices on an EC-labelled graph and must terminate the traversal if the loop is detected. To consider header transformation in the traversal, we treat encapsulated packet headers as a stack of packet headers. Since packet forwarding behavior is deterministic, the same contents and orders of the header stack will always appear periodically, or the stack will always be encapsulated with the same (stack of) header at some point if there is a loop. Thus, the loop condition on the EC-labelled graph is to either (1) traverse a vertex using the same EC-label stack that has been used in the previous traversal or (2) traverse a vertex

Algorithm 4: Infinite loop check

```
   Input: device name src
   Output: Boolean indicating whether a loop exists
1  Function LoopCheck(src):
2      VESize ← VtoECLStack[src].size
3      ESize ← ECLStack.size
4      if VESize ≥ ESize then
5          if VtoECLStackVec[src].find(ECLStack) then
6              return True
7      else
8          BStack ← GetStackBottom(ECLStack, VESize)
9          if VtoECLStack[src] = BStack then
10             TopSize ← ESize − VEize
11             TStack ← GetStackTop(ECLStack, TopSize)
12             if VtoECLStackVec[src].find(TStack) then
13                 return True
14             VtoECLStackVec[src].add(TStack)
15             return False
16     VtoECLStackVec[src].add(ECLStack)
17     return False
```

while stacking the same (stack of) EC labels that have been used in the previous traversal for the vertex (the proof is in §C.4).

We develop Algorithm 4 using a stack of EC labels (EC-label stack) to detect loops. The algorithm uses three types of EC-label stacks. First, *ECLStack* stores the current EC labels obtained from the previous traversal. Second, the associative array *VtoECLStack* stores the EC-label stack for vertices used in their previous traversal. Third, the vector *VtoECLStackVec* stores the EC-label stacks given as input and has a function "find" to check whether it already contains the given stacks. We can access them as global variables. For loop detection, the algorithm first checks if the stack size in the previous traversal is greater than the current stack size. If so, it checks whether the current stack has been used in the previous traversal for loop condition (1) (L 4 to 6). Otherwise, the algorithm checks if there is a loop induced by the encapsulation with the same (stack of) labels for loop condition (2) (L 7 to 15). It creates a new stack (*BStack*) by extracting contents from the bottom of ECLStack up to *VESize* (L 8) and checks if BStack has been used as in the previous traversal (L 9). If so, it next extracts contents from the top of ECLStack up to *TopSize* and checks whether the stack consisting of the contents (*TStack*) has been used in the previous (L 10 to 12).

With Algorithm 4, Algorithm 5 enumerates the transition relations by traversing an EC-labelled graph with header transformation rules. For each header (*htr.key*) transformed by the rules, it first gets a set of headers overlapping the transformed header by traversing the radix tree (L 25 to 27). Note that we use *tr.SrcHdr* in Algorithm 6 (L 29). For each overlapping header, it calls *RecTrace* to trace the label transformations (L 33).

The function *RecTrace* uses a depth-first search (DFS) to trace the label transformations in packet forwarding. The algorithm first checks whether there is a loop using Algorithm 4 (L 2 to 4). It then processes the header in accordance with a type of header transformation (L 5 to 18). In the case of a packet encapsulation, it pushes the current header into *Stack* and the current EC label into *ECLStack* and gets an outer header *NewHdr* (L 5 to 8). In the case of a packet decapsulation, it pops the encapsulated header from *Stack* and the top

Algorithm 5: Tracing the transformations

```
   Input: a set of transformation rules Htr, a radix tree T
   Output: a set of transition relations Tr
1  Function Trace(Htr, T):
2      for htr in Htr do
3          src ← htr.device
4          Hdr ← GetOverlappingHdr(htr.key, T)
5          for hdr in Hdr do
6              tr.SrcHdr ← hdr
7              ECLStack, VtoECLStack ← New
8              VtoECLStackVec ← New
9              ECLStack.Push(GetECL(hdr, T))
10             RecTrace(tr, src, hdr, Stack, T)
11             Tr.Add(tr)
12     return Tr
13 Function RecTrace(tr, src, hdr, Stack, T):
14     if LookCheck(src) then
15         report ForwardingLoop
16         return
17     while src.HT(hdr) ≠ null do
18         if src.HT(hdr).type = Encap then
19             Stack.Push(hdr)
20             NewHdr ← src.HT(hdr)
21             ECLStack.Push(GetECL(NewHdr, T))
22             tr.add(src, GetECL(hdr, T), Encap, GetECL(NewHdr, T))
23         else if src.HT(hdr).type = Decap then
24             if Stack = null then return
25             NewHdr ← Stack.Pop()
26             ECLStack.Pop()
27             tr.add(src, GetECL(hdr, T), Decap, GetECL(NewHdr, T))
28         else if src.HT(hdr).type = Rewrite then
29             NewHdr ← src.HT(hdr)
30             ECLStack.Swap(GetECL(NewHdr, T))
31             tr.add(src, GetECL(hdr, T), Rewrite, GetECL(NewHdr, T))
32         hdr ← NewHdr
33     VtoECLStack[src] ← ECLStack
34     for s in GetNexthop(G, src, GetECL(hdr, T)) do
35         RecTrace(tr, s, hdr, Stack, T)
```

of the label from *ECLStack* (L 10 to 13). In the case of a packet rewrite, it transforms the current header into a new header and swaps the top of the label on *ECLStack* (L 15 to 17). The algorithm then models the header transformation as a transition relation of EC labels (L 9, 14, and 18). Next, it records the current EC-label stack *ECLStack* to the EC-label stack of the vertex *VtoECLStack[src]* (L 21). *RecTrace* is recursively called for tracing further EC label transformations if there are some nexthop from *src* on G (L 22 and 23). Note that we can access the graph G as a global variable (L 22).

Encoding the relations as a hash

Algorithm 6 encodes the transition relations computed by Algorithm 5 into a hash function to quickly handle header transformations in traversing an EC-labelled graph. For identifying header spaces of ECs that are transformed, the algorithm uses a unique label called "transition label" because some headers in the EC may not be transformed (L 2). The algorithm adds

Algorithm 6: Encoding as a hash function

```
   Input: a set of transition relations Tr, a radix tree T
   Output: a hash function Hs
1  for tr in Tr do
2      TrLabel ← GetNewTrLabel()
3      AddTrLabelToRadix(TrLabel, T, tr.SrcHdr)
4      for tr_i in tr do
5          Hs[TrLabel][tr_i.srcECL][tr_i.S] ← < tr_i.trECL, tr_i.Type >
6  return Hs
```

a transition label to a leaf of a transformed header (*tr.SrcHdr*) of the radix tree (L 3). Afterward, it encodes each transition relation as a hash function *Hs* (L 4 and 5). We use the hash function to transform EC labels in graph traversal. Figure 5.4 shows the history when we traverse the network in Figure 5.2. We assume that the transition relations in Figure 5.4 are encoded into the hash function with transition label 1.

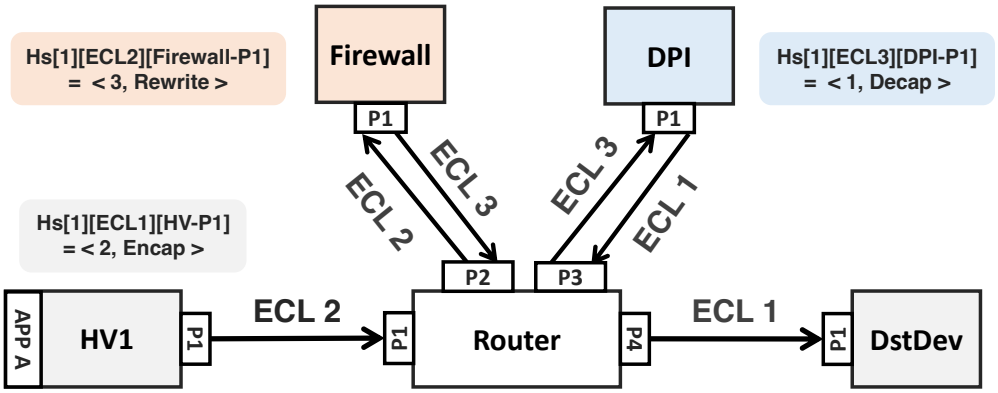
5.4.4 Verification algorithm

Graft supports both static and incremental verification. For realtime verification, we incrementally verify whether the current forwarding behavior satisfies the operator's policies given a data plane update. There are two steps for the verification: (1) computing which ECs are affected by the update and (2) traversing an EC-labelled graph using the affected EC labels and transition labels to verify the policies. Since we manage header spaces of ECs with a radix tree, we find out the affected ECs by traversing the radix tree. For instance, when a new forwarding rule is added, we traverse the tree on the basis of the forwarding prefix, find out the affected headers in the tree, and get their EC labels and transition labels, if they exist, from the leaves. We then add the new edge to the forwarding paths of the affected ECs on the EC-labelled graph. Finally, we check policy satisfaction by traversing the paths. We use the hash function computed by Algorithm 6 in the traversal to consider header transformations. As described in §5.4.2, Graft can verify several forwarding properties such as reachability, blackhole, forwarding loop, hop limit, waypoint, and isolation. These verification algorithms are based on the general DFS-based graph search algorithm (ReachDFS) on our EC-labelled graph with the infinite loop check (Alg. 2), and we do not claim novelty. Thus, we provide the search algorithm and its soundness in §C.2 to §C.3. Graft can similarly verify other properties by modifying or extending the algorithm.

5.5 Implementation

We implemented Graft in 3.5K lines of C++ code. We currently implemented a parser for data plane rules, such as forwarding and header transformation rules (e.g., SRv6) for Cumulus Linux, Arista Networks, and FRRouting (FRR) [180].

Path representation of ECs. To efficiently manage string-based forwarding paths of ECs in Algorithm 3, we represent them as a string of numbers, not a string of devices and their ports. We assign unique numbers called Link IDs to each directed edge on an EC-labelled graph. Thus, the forwarding paths of ECs are represented as a sequence of Link



Traversed Edges	ECL for forwarding	EC-label Stack
HV1-P1 to Router-P1	2	1:2
Router-P2 to Firewall-P1	2	1:2
Firewall-P1 to Router-P2	3	1:3
Router-P3 to DPI-P1	3	1:3
DPI-P1 to Router-P3	1	1
Router-P4 to DstDev-P1	1	1

Figure 5.4: History of the traversal from HV-P1 to DstDev-P1 on the EC-labelled graph in Figure 5.2.

IDs. This technique reduces the total length of each forwarding path, which improves memory usage and verification time.

Device model and its API. In Graft, a network device is modeled with a set of data plane functions and data structures for packet forwarding. To easily model realistic packet processing pipelines, we provide an API for network operators to model each network device by connecting data plane functions as modules. This modularity enables network operators to easily model different packet processing pipelines in network devices. In addition to the modular device model, we provide an API for the operators to define a header format for header transformation protocols. There are multiple network protocols and header formats for header transformations. Thus, we need a user-defined API for handling them in verification. Algorithm 5 and Algorithm 6, used for modeling and embedding header transformations into an EC-labelled graph, work in accordance with the defined header formats through the API.

5.6 Evaluation in emulation networks

For scalability evaluation, we show that Graft outperforms a publicly available BDD-based data plane verifier [42]. We compare the verification time and memory usage of Graft and APT, which is provided as an open-source implementation in Java.[5] We first evaluate their performance in three scenarios: the processing time of their data plane models (§5.6.1), that for incremental verification by a data plane update (§5.6.2), and that for a what-if analysis with link failures (§5.6.3). In addition, we discuss their memory usage (§5.6.5). Second, we show that Graft handles header transformations with low overhead for scalability. We

[5]Katra is not available publicly at the time of writing. We discuss the qualitative comparison of Graft and Katra in §5.8.

Table 5.1: Synthesized network datasets

#Router	#ToR	#Leaf	#Spine	#Link	#Fwd. rule
100	64	32	4	2304	8.63×10^5
200	128	64	8	8704	3.74×10^6
300	192	96	12	11019	8.95×10^6
400	256	128	16	13326	16.74×10^6
500	320	160	20	15633	35.02×10^6

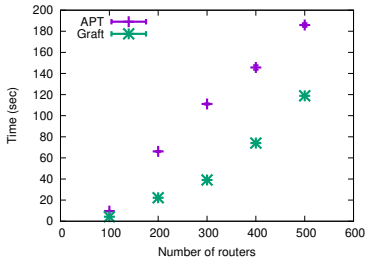


Figure 5.5: Processing time for creating data plane model.

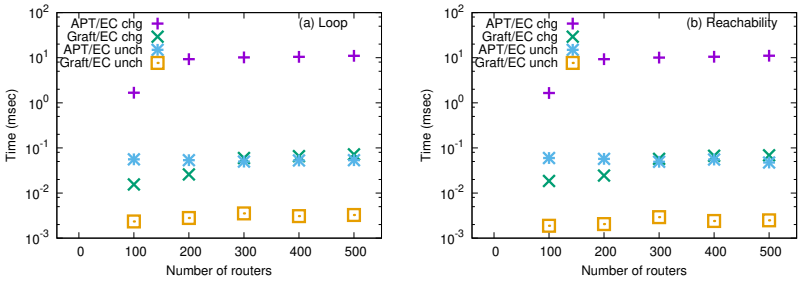


Figure 5.6: Average verification time for data plane update.

measure the processing overhead of the header transformations (§5.6.4) and its memory overhead (§5.6.5). We run all of the following experiments on a single core on a commodity server (2.70 GHz Intel Xeon processor with 28 CPU cores/128 GB RAM). As the large-scale synthetic network datasets, we emulate five 3-tier Clos networks [159] with varying numbers of routers and forwarding rules using Containerlab [181] since today's big cloud vendors adopt the Clos networks in their datacenter [10,11,175]. Table 5.1 shows the dataset overview.

5.6.1 Creating data plane model

We first measure the processing time for creating a data plane model with Graft and APT. The processing time consists of computing ECs (§5.4.1) and embedding the forwarding behavior of each EC on the data plane model (§5.4.2).

Figure 5.5 shows the average time for the processing. We observe that the computing time of Graft is 1.5x to 3.3x faster than APT for 200-500 routers. The difference is due to how Graft and APT compute ECs. APT first computes predicates that specify how packet headers are processed on each network device using forwarding rules. The forwarding predicates are represented by BDDs. APT then creates atomic predicates that specify minimal sets of ECs using these predicates. These processes take a relatively long time due to the operation to compress its internal data representation. In contrast, Graft computes path-based ECs by leveraging packet header structures in Algorithm 3 to avoid the BDD's overhead of header space management.

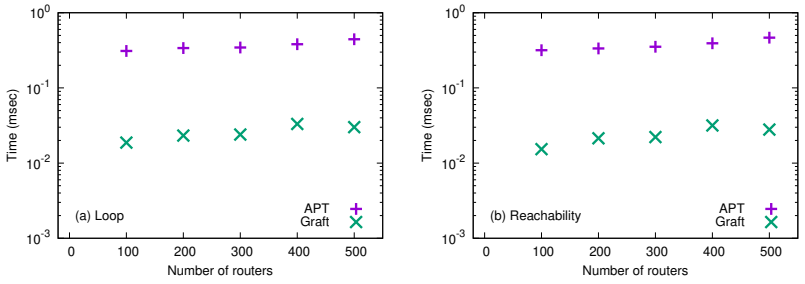


Figure 5.7: Total verification time for answering what-if queries.

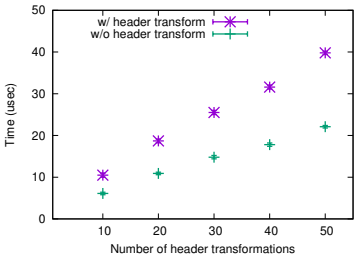


Figure 5.8: Processing time for handling header transformations.

5.6.2 Incremental verification by an update

We next evaluate the incremental verification time of Graft and APT for a data plane update. In this experiment, we use two data plane update scenarios. The first scenario is a data plane update that changes existing ECs. Graft and APT have two tasks: updating the ECs and verifying the network properties. The second one is an update that does not change the existing ECs, i.e., Graft and APT only verify the network properties for the update without updating the existing ECs. We measure the time to verify end-to-end reachability between a pair of network devices and a forwarding loop as network properties after adding a new forwarding route.

Figure 5.6 shows the average verification time of Graft and APT. In the first scenario (Graft/EC chg and APT/EC chg), we see that the verification time of Graft for the forwarding loop (a) and the reachability (b) is 89x to 380x faster than APT. In the second scenario (Graft/EC unchg and APT/EC unchg), the time of Graft is 14x to 100x faster than APT. The main reason for the improvement in the first scenario is that Graft avoids BDD's overhead in updating existing ECs. Since APT relies on BDDs to manage header spaces of the ECs, it must operate to compress its internal data representation every time the data plane update affects the ECs. Our measurement shows that the update time takes tens of milliseconds. In comparison, Graft can quickly find the ECs that the update affects by only traversing a radix tree. This process does not need any compression, and the overall verification takes only tens of microseconds. The improvement in the second scenario is due to how to check whether existing ECs are affected by the update. APT checks if a port predicate is changed for the update, which takes tens to hundreds of microseconds. On the other hand, Graft checks if the update affects the ECs by traversing the radix tree, which takes a few microseconds.

5.6.3 What-if analysis: Link failures

We evaluate the verification time for Graft and APT to analyze what-if queries from network operators. In the what-if analysis, we check whether the packet forwarding behavior satisfies the operator's policy if some links are down. We choose end-to-end reachability and forwarding loop detection as the policy. We measure the total time to randomly cut the links and verify the policy for every link cut. Note that the what-if analysis does not need any EC updates.

Figure 5.7 shows the total verification time of Graft and APT for random link cuts

five times. In both the forwarding loop (a) and the reachability (b), we see that the total verification time of Graft is 10x faster than APT. The reason is that Graft traverses the edge-labelled graph using EC labels, but APT operates intersection to represent the header sets in the traversal. Thus, Graft is more suitable for handling burst updates in link failures (e.g., thousands of data plane updates within a second [48]).

5.6.4 Overhead of header transformation

Here, we focus on the overhead of Graft in handling header transformations. To measure the overhead, we use two scenarios using a chain topology. One is a network that does not transform a packet header. The network sends a packet from a source to a destination. The other is a network that transforms a packet at every hop. The first half of routers encapsulate the packet in an IP header, and the second half decapsulates the packet. For instance, the packet is encapsulated in the IP header five times and decapsulated five times for a 10-hops network. We compare the verification time to check the reachability between the source and destination routers on the two networks.

Figure 5.8 shows the two verification times. We see that Graft handles header transformations in tens of microseconds. The reason for this low overhead is that we use only the hash function to handle the header transformations *Hs* in traversing the EC-labelled graph. Our result shows that Graft efficiently handles the header transformations for sub-millisecond level verification.

5.6.5 Memory usage

We compare the memory usage of Graft and APT in verification. We observe that Graft takes 6.7 GB of physical memory for 500 routers while APT takes 5.8 GB, which is a trivial difference.

Next, we measure Graft's memory overhead by varying the number of header transformation rules in Clos (500 routers). Considering realistic settings, we assume all ToR switches house 20 servers, each with 10 to 50 VMs (applications) on a hypervisor. In addition, each hypervisor on the servers has a header transformation rule for each VM. For instance, for 10 VMs in each server, the total number of header transformation rules is 64K. Through varying the transformation rules, we observe that the required memory size only increases by one percent (from 6.72 GB to 6.83 GB) when installing 320k transformation rules (50 VMs). This result shows that Graft can handle a varying number of header transformation rules with low memory overhead for scalability.

5.7 Evaluation in a production DCN

We evaluate Graft on a real production DCN [172,175]. For scalability in real-world large-scale networks, we first measure and compare the processing time of their data plane models and that for incremental verification of forwarding loops with Graft and APT (§5.7.2). Second, we demonstrate that Graft is expressive for header transformations by modeling/verifying real SRv6-based SFCs in Figure 5.1 (§5.7.3). Note that the SFCs in Figure 5.1 are simplified

for an explanation, but we use the non-simplified real SFCs in the evaluation. Finally, we show that Graft can verify a distributed NAT system [172] and detect the cause of an actual network failure (blackhole). We run all experiments on a single core on a commodity server (a 2.20 GHz Intel Xeon processor with 40 CPU cores/62 GB RAM).

5.7.1 Dataset description

The DCN is a 3-tier Clos network. We create datasets with the network operators by collecting forwarding rules from routers and hypervisors in the network with its topology. We use one of the tenants of the whole DCN in this evaluation. The dataset for scalability evaluation (§5.7.2) contains 1700 hypervisors and 596 routers with 8.55×10^5 forwarding rules. The dataset for expressiveness evaluation (§5.7.3) contains 291 hypervisors and 82 routers with 4.55×10^6 forwarding rules. The dataset for a distributed NAT system [172] contains 68 NAT servers with 1.16×10^6 lines of NAT/IPIP rules. In these networks, all routes are advertised by eBGP, and traffic is equally forwarded among multiple paths using Equal Cost Multipath (ECMP). In addition, route summarization like [10] is used to minimize FIB sizes. Thus, the number of forwarding rules in the networks is relatively small compared to the network scale.

5.7.2 Scalability evaluation

We first measure the time of building data plane models with Graft and APT. We observe that APT takes 22.46 minutes and Graft takes 4.28 seconds for the model build, confirming that Graft is 314x faster than APT. We see that APT takes over 21 minutes to compute atomic predicates. In short, APT takes most of the building time to compress internal data representation. On the other hand, Graft computes ECs in a few seconds. There are two reasons for the low overhead. First, unlike BDD, Graft does not need to compress its internal representation since it leverages header structures in EC computation with a radix tree and a hash table. Second, Graft can leverage the locality of network prefixes. Since routes in the network are summarized, leveraging the radix tree naturally considers the locality, which reduces the path computation time between overlapping prefixes.

Next, we measure the time of checking the forwarding loop at a data plane update. We observe that APT takes 7.41 seconds and Graft takes 352.3 microseconds for the loop check, confirming that Graft is 21027x faster than APT. Similar to the result in §5.6.2, the main reason for the difference is that Graft avoids the compression overhead in updating ECs. We observe that APT takes over 6 seconds for compression in this measurement. On the other hand, Graft takes several hundred microseconds to update ECs, leveraging a radix tree and hash-based EC management. In summary, Graft achieves realtime forwarding loop detection on the large production network dataset.

5.7.3 Modeling and verifying SFC with SRv6

For expressiveness, we demonstrate how Graft verifies the forwarding policy in the production SFCs [175]. The SFCs are a good example to show the expressiveness of Graft because it uses all primitive packet processing of header transformations for sophisticated network techniques. Here, we suppose that a network operator verifies packet reachability from App A to the

Internet through the SFC (Firewall and DPI). As a failure scenario, we consider a case in which the operator sets a wrong configuration in HV2, which causes packet drops between HV2 and DPI.

We take three steps for the verification. First, we create packet processing pipelines for each network device model by connecting data plane functions using a simple API provided by Graft. For instance, we create a pipeline on HV1 in Figure 5.1 as follows. Note that while SRv6 has multiple functions for encapsulation (e.g., T.Encaps), we here describe "SRv6_Encap" for simplicity.

```
HV1Pipeline = ["SRv6_Encap", "Forward"];

SetPktProcPipeline("HV1", HV1Pipeline);
```

`SRv6_Encap` module handles SRv6 encapsulation for packets with specific source addresses (for App A). `Forward` module is for IP packet forwarding. These two modules are installed in a network device model of HV1. Second, we set the reachability policy for the verification using a Graft's API:

```
AddReachabilityPolicy("App_A","Internet");
```

After these configurations, Graft creates an EC-labelled graph with forwarding rules.

When the operator sets a misconfiguration in HV2, Graft verifies the reachability policy in 228.2 microseconds. In the verification, Graft provides where the policy violation occurs to the operator (the output is "Vertex: HV2-P1, ECL: 3, Func module: Forward"). This case study shows that Graft is expressive to verify sophisticated network techniques in production DCNs.

5.7.4 Detecting blackhole in distributed NATs

Finally, we demonstrate a real case study in a distributed NAT system in the DCN [172] to show the effectiveness of Graft. This system consists of 68 NAT servers for distributed processing. Each NAT has a specific range of source addresses and rewrites the source address of an incoming packet header if the address belongs to the range. Otherwise, the NAT redirects the packet to another NAT responsible for the rewriting using IP in IP (IPIP) [164].

In the network, a blackhole happens due to the update failure of IPIP redirecting rules. To continuously provide the NAT service, even if some of the NAT servers are down, the system has an automatic delegation mechanism for the address translation. When a NAT server is down, the system first delegates the range of the address translation in the NAT to the other working NATs. It then updates the redirecting rules in all NATs to send the packets belonging to the ranges to the delegated NATs. However, updating the redirecting rules sometimes fails due to the high CPU utilization of NAT servers, which redirect the packets to a failed NAT.

To identify the blackhole and its cause, we model the network with 1.16×10^6 lines of actual NAT/IPIP rules. Graft was able to model the network, verify the blackhole, and find out the cause in 30 sec [6] while the network operators did not notice it for over 1 month, and

[6]The main overhead is due to a JSON library for reading IPIP/NAT rules.

took three hours to fix it. This use case demonstrates the effectiveness of Graft in fixing real network failures.

5.8 Discussion

Evaluation datasets. Most of the data plane verification research uses open-source datasets such as Internet2 [182] and SDN datasets [183] for evaluation [41–45,47,50,58]. On the other hand, evaluation using production large-scale networks reveals real issues in their production environment, such as lack of scalability and expressiveness of their models (§5.7) and consistency of the verification results [26,48]. In this work, we show that both scalability and expressiveness for header transformations are inevitable to verify complex forwarding behavior in today's production DCNs.

Comparison of Graft and Katra. Katra handles header transformations with formal semantics [44]. While Graft focuses on selected header fields essential for packet forwarding, Katra theoretically supports infinite sequences and fields of packet headers in verification. On the other hand, Graft has a performance advantage against Katra by developing header space management leveraging header structures. For using BDDs, Katra implements several optimizations. However, these techniques are intended to quickly find overlapping ECs during graph traversal in verification, not to compute and manage header spaces and forwarding paths of ECs. From the results of our experiments (§5.7.2, §5.6.1, and §5.6.2), we found that the fundamental bottleneck for scalability is BDD-based header space management. Thus, Katra has inherited the same issues, as also pointed out in past literature [46,89].

Stateful network functions. Currently, Graft does not support stateful network functions such as stateful load balancer and NAT on DCNs. While stateful function verification [87,88,170] techniques are beneficial, they are not suitable for realtime verification in large-scale networks that house thousands of devices due to their scalability issues.

Nondeterministic processing. As with prior works, Graft supports nondeterministic forwarding and transformation by enumerating possible results. For instance, Graft handles ECMP by enumerating possible forwarding paths in the evaluation (§5.7). We have confirmed that such enumeration is not costly, even in the production DCN scale in the evaluation. If we know the settings of nondeterministic processing, we can incorporate the processing into our data plane model.

Consistency of data plane snapshots. Getting consistent snapshots of data plane is challenging for data plane verification [26,48]. Graft focuses on how to efficiently verify the operator's policies given the snapshots, not on how to get them. It could be helpful for operators to apply the techniques accurately, getting the snapshots [26,48] for Graft.

Coral [184] and Tulkun [177] address the consistency problem by verifying the data plane in a distributed manner. They use virtual instances (e.g., VM) for each device to get data plane conditions from the device and exchange them with other instances for verification. While Graft does not support such distributed verification, it does not need to provision instances per network device, which could be difficult in large-scale networks.

5.9 Summary

This research proposed Graft, a new data plane verification framework that simultaneously achieves expressiveness for header transformations and scalability of verifying complex packet forwarding in large-scale networks. For scalability, we proposed EC computation and management methods leveraging the longest-prefix match using a radix tree and hash tables. For modeling packet forwarding with header transformations, we proposed a data plane model and algorithms that treat the transformations as transition relations of ECs. Through our evaluation using synthetic/production DCNs, we showed that Graft achieves both scalability for verifying large-scale networks in realtime and expressiveness enough to model the transformations. Besides, we demonstrated its effectiveness in fixing network failures by detecting the cause of a real failure in the production DCN.

Chapter 6

Discussion and Limitation

The three works in this dissertation contribute to improve the practicality of network verification and configuration repair techniques by addressing two important problems: (1) the difficulty of writing specifications for verification/repair and (2) the insufficient expressiveness of network models for sophisticated network functions (e.g., SFC and TE). The first two works (chapter 3 and chapter 4) lower the barrier to leveraging repair and verification techniques by reducing the burden of their specification writing. Specifically, they enable network operators to safely automate configuration repair by preventing unintended side effects and to analyze network failures using symbolic queries rather than complete specifications. These advancements make it more feasible to leverage the repair and verification techniques in daily network operations.

The third work (chapter 5) develop a new data plane verification framework that supports header transformations, which are primitive packet processing for modeling the sophisticated functions and (2) is scalable for large-scale networks ($>$ 2000 network devices). By simultaneously achieving the model expressiveness with scalability, this work enables network operators to verify complex packet forwarding behaviors in today's large scale networks (e.g., data center networks). Furthermore, because the model represents packet forwarding behaviors on data plane as a geneal graph structure, future control plane verification, data plane verification, and configuration repair techniques can leverage it to model sophisticated network functions.

In this section, we present a discussion associated with these works and their limitations when applying them to real-world network operations.

6.1 Discussion

Collecting configurations and data plane updates. Our three techniques rely on router configurations, forwarding tables, and the network topology to construct an accurate network model. In our real-world case studies, we collected these necessary configurations and forwarding rules from each router via a dedicated management network. However, if any of this data is missing or incomplete, a discrepancy arises between the constructed model and the actual network state. Consequently, the results of verification or repair may become inconsistent with the real-world network behavior. To effectively leverage these techniques,

we need a reliable mechanism for collecting all required network data.

The order of verified/repaired configuration updates. When applying verified and repaired configurations to networks, we should consider the order in which they are installed on routers [123]. The networks exhibit different packet forwarding behaviors during their convergence phase (i.e., before routes converge) compared to their final, stable state. Existing verification and repair tools typically focus on the correctness of the final state but do not prescribe the specific sequence for these configuration updates. Therefore, careful planning is required for the deployment order to avoid transient inconsistencies or forwarding loops. The use of specialized techniques dedicated to ensuring consistent network updates, such as those proposed in [123], is highly recommended.

Leveraging LLM for repair and verification. Leveraging LLM for repair and verification is promising to efficiently solve the computationally expensive problems (e.g., verification [83] and repair [133]). However, LLMs often suffer from issues such as hallucinations and a lack of formal guarantees (as we also discussed in §3.9). In contrast, the three works presented in this dissertation are based on deterministic systems and algorithms. Therefore, these works can complement LLM-based approaches by providing the rigorous validation and deterministic execution that LLMs currently lack. Developing such a hybrid approach remains an important direction for future work.

6.2 Limitation

Supported network properties. The three techniques presented in this work focus on static network properties—such as reachability, waypoint enforcement, and loop freedom—that can be derived from router configurations and packet forwarding rules. Conversely, they do not support the analysis of dynamic network properties, including packet drop rates or latency guarantees that are important for ensuring service-level agreement (SLA) [154–156]. This limitation stems from the fact that our models are constructed exclusively from static configurations and forwarding rules, which do not capture the real-time state (e.g., traffic load, queue depths) required to simulate and model such dynamic behaviors. Probabilistic network verification [40,185,186] or performance verification [60,187,188] are useful techniques to verify the dynamic properties because of their models.

Specification type for debugging, verification, and repair. The specifications used in our three techniques are defined as forwarding path requirements based on regular expressions. While effective for path-based properties, this representation makes it inherently difficult to specify behaviors involving temporal constraints. Overcoming this limitation would require adopting specifications based on formalisms capable of reasoning about time, such as temporal logic. This would also require the development of new verification models and algorithms to check network properties against temporally aware specifications.

Scalability for hyper-scale networks. While we demonstrated that each technique achieved sufficient scalability, these evaluations were conducted on networks comprising at most several thousand network devices (e.g., routers and switches). However, today's data center networks can reach significantly larger scales, potentially involving 10,000 (10K) switches or 1000M routes [81,101]. We do not evaluate the scalability of our proposed techniques within such hyper-scale environments in this dissertation. A possible extension of

our techniques for this scale is to leverage the distributed and/or incremental verification approaches [36,81,189].

Software and hardware bugs. Our three techniques cannot identify software and hardware bugs that can not be inferred from router configurations and forwarding rules. This is because the internals of network devices (e.g., operating systems and ASICs) are black-box for network verifier and repair tools; therefore, these tools can not create network models that consider the bugs.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Router misconfigurations can trigger severe network failures and cause significant downtime, disrupting critical infrastructure in our daily lives. Configuration repair and network verification offer a promising approach to prevent such failures before deployment and pinpoint their root causes when they occur. While some large cloud providers have begun leveraging these techniques, they are not yet widely adopted in operational networks.

This dissertation presented three techniques to address the two key issues for the practicality of the verification and repair techniques toward their real-world deployment.

Reducing the difficulty of writing correct specifications. The first two techniques tackled the difficulty of correctly writing specifications for repair and verification. For configuration repair, we proposed a specification refinement approach that guides repair tools to satisfy initial network change intents without introducing undesired side effects via repair specifications. This approach reduced the operator's burden of writing correct specifications to the two essential tasks: (1) defining new change intents and (2) determining the priority between the change intents and unavoidable side effects. To achieve this, we developed ISR, an augmentative framework for repair tools that performs the specification refinement through side effect diagnosis. We demonstrated through OSPF and BGP configuration repair that ISR guided three different repair tools (SMT-solver-based, SMT/Simulation-based, and LLM-based repair tools) to satisfy new change intents without undesired side effects. Furthermore, ISR completed the diagnosis within ten seconds on several real-world networks and their conditions.

In the second technique, we proposed a symbolic extraction approach of packet forwarding behaviors without complete specifications, such as data plane verification. We developed Lupe, a system that enables operators to (1) specify symbolic packet headers and/or forwarding paths and (2) obtain all packet forwarding behaviors matching the symbolic values for identifying the failure causes and impacts. We showed that Lupe extracts forwarding behaviors matching given symbolic headers or paths from all possible behaviors in 15 seconds, even in the large-scale data center network. This represents a 5x-129x improvement over adapting existing algorithms from data plane verification. We also demonstrate that Lupe helps operators identify the causes and impacts of network failures in real-world networks.

Improving the expressiveness of data plane models. The third technique improved the model expressiveness to widely apply data plane verification techniques for real-world operation. To this end, we extended existing data plane models (packet forwarding behavior models) for sophisticated network techniques such as service function chaining and traffic engineering with scalability for large-scale networks. We proposed Graft, a new data plane verification framework that simultaneously achieves expressiveness and scalability. Graft has a formal network model to support the sophisticated techniques with efficient algorithms and data structures for forwarding behaviors in large-scale networks. Our comprehensive evaluation, using both synthetic and production DCNs, confirmed that Graft meets both requirements. Graft verified several network properties 100x faster than the existing approach in the synthetic DCNs and 20000x faster in the production DCN. Second, we demonstrated its expressiveness by verifying forwarding behaviors with the sophisticated techniques in production data center networks.

7.2 Future work

Explainability of verification and repair. Enhancing the explainability of verification and repair processes remains an important open problem for practical deployment. While both techniques can identify failures and generate correct configurations, they often provide what (e.g., a counterexample packet) or how (e.g., a configuration diff) but not why (e.g., why verification/repair failed).

A potential future direction focuses on improving the explainability of diagnosed failures and suggested repairs. For verification, this involves moving beyond simple counterexamples to pinpoint the root cause of the failure, such as identifying the precise configuration lines or specific forwarding rules responsible for the undesired behavior. For repair, the system should not only provide a fix but also justify its reasoning. This explanation should clarify why the suggested change is correct, why it is preferable to other potential fixes, and explicitly confirm how it avoids the side effects diagnosed during the process. Integrating such explainability is crucial for building operators' trust, reducing the cognitive load on them in real-world operation.

Automatic generation of high-quality specification. While our proposed techniques reduce the operational burden of specification writing, human effort is still required. A valuable future direction is to address the automatic generation of high-quality specifications from more human-friendly, high-level formats. One promising approach is to leverage recent advances in natural language processing (NLP), including Large Language Models (LLMs), to directly translate operator intents expressed in natural language into the formal specifications. However, current LLMs suffer from well-known accuracy issues, such as misinterpreting ambiguous operator intent or failing to generate syntactically perfect formal output [100,103]. Therefore, balancing the generative efficiency of LLMs with the rigorous accuracy demanded by formal methods remains a significant research challenge for automatic specification generation.

Heuristic techniques for the computational complexity of configuration synthesis and repair. Configuration synthesis and repair have been known as computationally hard problems [133]. Therefore, it is crucial to consider approaches that leverage effective heuristics to practically circumvent these computational challenges. Addressing this challenge

is inevitable for applying synthesis and repair techniques to today's large-scale networks.

Appendix A

Omitted discussion and proof in chapter 3

A.1 Human-in-the-Loop Specification Refinement

While ISR creates specifications to prevent all detected side effects by default, we recognize that not all of them are harmful. In real-world operations, some side effects may be acceptable or unavoidable to achieve new network change intents. We adopt this default design because the criteria for determining whether a side effect is acceptable or not depend on network operators and/or specific network situations, making it difficult to define a universal definition for "acceptable" side effects in advance.

For more practical usecases, our approach can incorporate network operators into the iterative specification refinement process to assess whether detected side effects are acceptable. While it imposes the burden of checking side effects on the operators, accepting minor side effects offers two practical benefits: (1) reducing the likelihood of specification conflicts, and (2) reducing the total number of refinement iterations. We illustrate the overall workflow in Figure A.1. When ISR identifies side effects, an operator reviews them with the causality information provided by ISR:

- **Acceptance:** If the side effect is acceptable, the operator asks ISR to permit it. ISR then generates a new specification that describes the forwarding path classified as the side effect. As described in §3.3.2, ISR treats the new specifications as additional network change intents.

- **Rejection:** If the side effect is not acceptable, the operator asks ISR to prevent it. ISR then generates a new specification to preserve the original forwarding path prior to the side effect, as described in §3.3.1.

Crucially, this human-in-the-loop specification refinement preserves the termination guarantee of the process (§3.5). The key to this guarantee lies in the monotonic decrease of the number of *unconstrained forwarding paths* in each iteration. Regardless of whether an operator accepts or rejects a side effect, ISR generates a new specification to explicitly constrain the corresponding path behavior (either permitting the new path or requiring the original one). This action effectively removes the path from the set of unconstrained paths. Thus, the number of remaining unconstrained paths strictly decreases in every iteration, ensuring that the process terminates regardless of the operator's decisions.

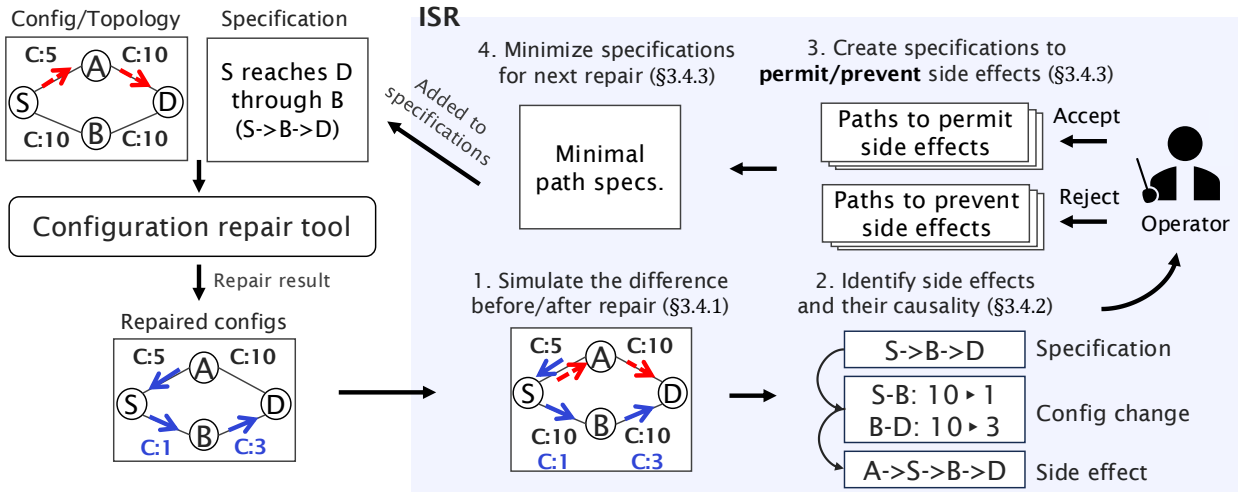


Figure A.1: Incorporating network operators into the iterative specification refinement.

A.2 Proof for Completeness and Soundness of Algorithm 1

Algorithm 1 correctly identifies the minimal containment path cover M because it guarantees the following correctness:

1. **Completeness:** Any path $m \in M$ (a maximal path) will never be in `CoveredPaths` before it is visited, as no path $q \neq m$ exists such that $m \sqsubseteq q$. Thus, m will always be added to M (Line 10).

2. **Soundness:** Any path $p \notin M$ (a non-maximal path) is, by definition, a subpath of some maximal path $m \in M$. Since length$(m) \geq$ length(p), the algorithm guarantees that m will be processed either before or at the same time as p. When m is added to M, p will be added to `CoveredPaths` (Line 15), preventing p from being incorrectly added to M.

A.3 Prompt Construction

In this section, we describe the prompts we created for NetBuddy in OSPF and BGP configuration repair (§3.6.1 and §3.6.2). In both repair, we modify and use the script to generate low-level configurations (generate_step_3_dataset.py) from the NetConfEval repository [126]. We run the gpt-oss:120b model as LLM for NetBuddy and do not use the Retrieval-Augmented Generation (RAG) mode.

A.3.1 Prompt for OSPF configuration repair

In addition to default system prompts used in the script, we provide the prompt illustrated in Figure A.2 to NetBuddy for the OSPF configuration repair (§3.6.1). This prompt instructs NetBuddy to act as a network engineer and update OSPF link costs to satisfy a given batch of path requirements. ISR automatically fills the placeholders {*current_costs*} and

```
You are a network engineer.
Your task is to update OSPF costs to satisfy a given batch of path requirements.
In OSPF, a cost is assigned to each direction of a link.
So, please consider the cost assignment problem by modeling the network as a graph.
Here are the current OSPF costs of all links in the network:

--- CURRENT COSTS ---
{current_costs}
--- END CURRENT COSTS ---

Your task is to identify which of the CURRENT COSTS need to be changed
to satisfy the following batch of requirements.
In OSPF, the lowest-cost path is preferred as the shortest path.
The cost of each link must be greater than zero.
Now, please satisfy all the following path requirements at the same time:

--- PATH REQUIREMENTS BATCH ---
{path_requirements}
--- END PATH REQUIREMENTS BATCH ---

You MUST output ONLY the links that require a cost change based on this batch.
Please minimize the path changes by your cost changes.
If some path requirements are conflict and can not be satisfied at the same time,
please report the conflict path requirements.
```

Figure A.2: Prompt for OSPF configuration repair (§3.6.1) using NetBuddy.

{*path_requirements*}. In the first iteration, these fields contain the initial link costs and the forwarding path requirement for the initial change intent. In later iterations, they contain the initial link costs and the updated (refined) forwarding path requirements from the previous repair results. In addition, the prompt defines the routing logic of OSPF where the lowest-cost path is preferred and constrains the output to list only the necessary link cost changes to minimize modifications, or to report conflicts if the requirements cannot be satisfied simultaneously. In response, NetBuddy returns a list of links and their updated costs to satisfy the given path requirements.

A.3.2 Prompt for BGP configuration repair

In addition to default system prompts used in the script, we provide the prompts shown in Figure A.3 to NetBuddy for the BGP configuration repair (§3.6.2). We create these prompts based on the scenario of the runtime BGP configuration modification in [116]. These prompts define the BGP configurations, such as AS numbers, prefixes, and BGP peers. They also provide the current forwarding path requirements and the change intent. For the second repair (Figure A.3b), we add a new forwarding path requirement to prevent the side effect found in the first repair. Specifically, this new requirement preserves the path from Customer 1 (router_as100) to Provider 1 (router_as20). Given the prompts, NetBuddy returns generated BGP configurations for each router in JSON format to satisfy the change intent.

I want to configure the network using BGP.
Please generate router configurations for router_as100, router_as20, router_as30, and router_as200 to satisfy the following intent.

Routers are located in the following ASes:
- router_as100 is in AS 100
- router_as20 is in AS 20
- router_as30 is in AS 30
- router_as200 is in AS 200

Here are the current router configurations:
- router_as20 advertises 20.0.0.0/16
- router_as200 advertises 200.0.0.0/16
- router_as20 and router_as30 are two providers with peering
- router_as200 is a customer of router_as30
- router_as100 is a customer of both router_as20 and router_as30 (multi-homed)
- router_as100 reduces the local preference attribute of the incoming BGP announcements sent by router_as30

Here are the current forwarding path requirements:
- **router_as100 has two paths to reach router_as200:**
(1) primary path (router_as100, router_as20, router_as30, router_as200) and
(2) backup path (router_as100, router_as30, router_as200)

Here are the intent for configuration change:
- Operators in AS 100 intend to switch the primary path of router_as100 from router_as20 to router_as30

(a) Prompt for first repair

I want to configure the network using BGP.
Please generate router configurations for router_as100, router_as20, router_as30, and router_as200 to satisfy the following intent.

Routers are located in the following ASes:
- router_as100 is in AS 100
- router_as20 is in AS 20
- router_as30 is in AS 30
- router_as200 is in AS 200

Here are the current router configurations:
- router_as20 advertises 20.0.0.0/16
- router_as200 advertises 200.0.0.0/16
- router_as20 and router_as30 are two providers with peering
- router_as200 is a customer of router_as30
- router_as100 is a customer of both router_as20 and router_as30 (multi-homed)
- router_as100 reduces the local preference attribute of the incoming BGP announcements sent by router_as30

Here are the current forwarding path requirements:
- **router_as100 has two paths to reach router_as200:**
(1) primary path (router_as100, router_as20, router_as30, router_as200) and
(2) backup path (router_as100, router_as30, router_as200)
- **router_as100 has a path to reach router_as20: path (router_as100, router_as20)**

Here are the intent for configuration change:
- Operators in AS 100 intend to switch the primary path of router_as100 from router_as20 to router_as30

(b) Prompt for second repair

Figure A.3: Prompt for BGP configuration repair (§3.6.2) using NetBuddy.

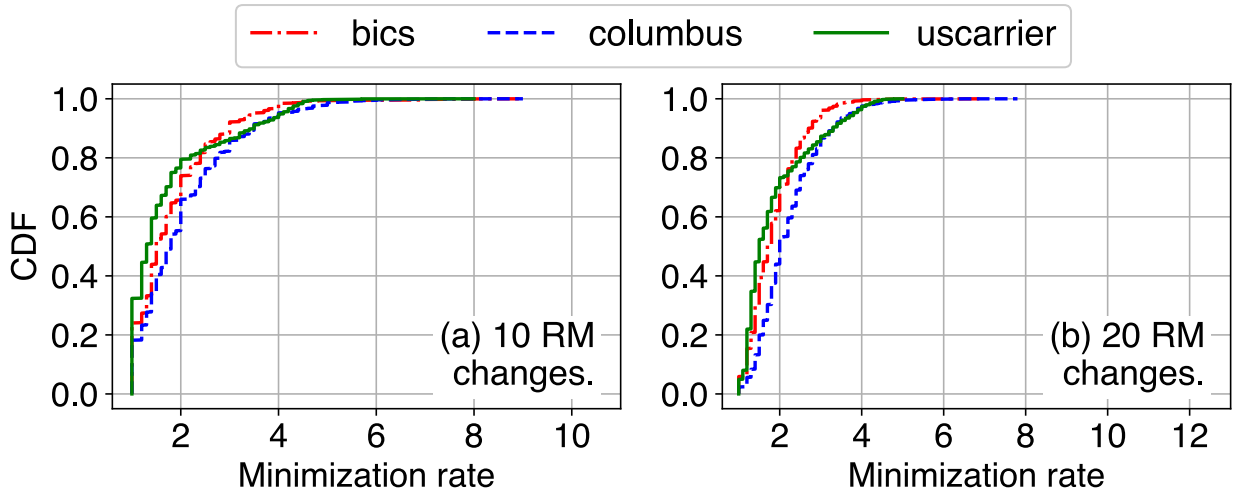


Figure A.4: Specification minimization for route map changes in BGP configurations. RM denotes Route Map.

A.4 Specification Minimization for Route Map Changes in BGP Configurations

We share the results of specification minimization rates for route map changes in BGP configurations used in §3.7.1. Figure A.4 presents the CDFs of the specification minimization for the route map changes. While the overall minimization rates are slightly lower than the parameter changes in §3.7.3, ISR still consistently achieves a reduction (rate $>$ 1.0). Consistent with the results in §3.7.3, the minimization rates remain topology-dependent, with UsCarrier (green line) continuing to underperform the other topologies. Furthermore, increasing the number of changes from 10 (a) to 20 (b) had little effect, consistent with the observation for local preference changes in §3.7.3.

A.5 Implementation detail of S^{2}Sim

We reproduce S^{2}Sim's route propagation graph (RPG) based repair algorithm for OSPF networks by following the formulation in the original paper [29]: we first simulate shortest-path routing on the given topology and link costs, and use the given path requirements to construct a RPG that annotates each (router, destination) pair with its desired forwarding path(s). We then encode each directed link cost as an integer variable in an SMT model and add constraints ensuring that, for every (router, destination), all RPG paths are shortest (and equal-cost in the multipath case) while any alternative simple path has strictly higher total cost. To mitigate side effects, we additionally minimize the total L1 distance between original and repaired link costs, i.e., we solve for a satisfying assignment that also minimizes the sum of absolute cost changes across all links.

NetComplete does not incorporate such a minimization objective; it simply searches for any valid link cost update that satisfies the constraints (e.g., assigning a cost of 1). Consequently, S^{2}Sim generates link cost updates that are less disruptive to unspecified paths compared to NetComplete (Table 3.1 in §3.6.1).

Appendix B

Omitted proof and algorithm in chapter 4

B.1 Correctness of computing all possible forwarding behaviors

Here, we prove that our header space computation based on lattice does not overlook any packet headers computed from forwarding rules given as input in Algorithm 2. To this end, we first define completeness, a notion to formally describe all possible packet headers.

Definition 12. (Completeness) A set of header spaces $\{HS_1, HS_2, ..., HS_n\}$ is complete if it satisfies the condition such that $H = HS_1 \cup HS_2 \cup ... \cup HS_n$ where H is the set of all packet headers in a network.

Theorem 3. *The lattice generated by Algorithm 2 covers a set of packet headers that satisfy the completeness in Definition Definition 12.*

Proof. Definition 3 requires that the union of the set of header spaces equals the set H that covers all packet headers in a network. We create subtrees as disjoint destination prefix groups that satisfy the following two conditions:

1. The destination prefix of a top node in each subtree includes destination prefixes of the rest of the other nodes in the subtree.
2. No destination prefix in a subtree overlaps with any destination prefix in other subtrees.

Algorithm 2 creates header spaces on lattices by each subtree. Lupe then merges the header spaces as a single space on a lattice. Therefore, we must prove that the union of header spaces represented by each lattice and the default route (the top node in the lattice) equals H. Since all packet headers that do not match any forwarding rule are forwarded by the default route, we can consider that the top node of the lattice includes these packet headers (i.e., headers that are not represented by the prefix groups). Then, we prove that each lattice completely covers all packet headers within their disjoint destination prefix ranges. Algorithm 2 checks whether a newly inserted header value and header values of existing nodes in each lattice have an inclusion or intersection relation (lines 19 to 31). If there is an inclusion relation, we put the node with the included header constraints beneath the node with the other value. If there is an intersection, the algorithm creates a new child node with the intersected header

constraints (line 28). Thanks to the property of lattice that is closed under intersection, the lattices create nodes with all header constraints generated by the intersection [147]. This fact shows that each lattice creates all packet headers within their destination prefix ranges. Therefore, we can conclude that the single header space merging each lattice equals H. □

Algorithm 7: Header space search on lattice

```
   Input: Query q, Lattice lattice
   Output: Packet deaders matching the query q
1  Function BfsOnLattice(q, lattice):
2      sublattices ← select sub-lattices related to the destination prefix in q from lattice
3      for all sublattice in sublattices do
4          RecBfsOnLattice(q, sublattice)
5  Function RecBfsOnLattice(q, sublattice):
6      stack ← new
7      for all parent in sublattice.bottom do
8          stack.push(parent)
9      while stack ≠ empty do
10         node ← stack.pop()
11         if is_visited[node] == true then
12             Continue
13         is_visited[node] = true
14         if node == NULL || node == sublattice.top then
15             Continue
16         if node.hdr ⊇ q.hdr then
17             Record header of cur.node
18         else
19             if node.hdr ∩ q.hdr then
20                 Record header of cur.node
21             for all new_node in node.parent do
22                 stack.push(new_node)
```

B.2 Algorithm for processing header query

We present a BFS-based algorithm on lattice to extract packet headers matching given header queries in Algorithm 7. There are two processes: (1) selecting the sub-lattices of the lattice related to the destination prefix of a query (lines 1 to 4) and (2) searching the sub-lattices by checking if header constraints between the query and the nodes in the sub-lattices have intersection and inclusion relations (lines 5 to 33). In the search process, the algorithm first checks if the constraint of the currently checked node in the sub-lattice includes the query's constraint (line 16). If so, it records the header constraint and terminates the subsequent searches (i.e., does not push its parents in the stack). This is because we concatenate the forwarding paths of the parents with the children's paths, so they already appear in the children (§4.4.1). Second, the algorithm checks if the constraint of the current node intersects

with the query's (line 19). If so, it records the node's constraint (line 20). Then, it continues the subsequent searches from the parents of the nodes (lines 21 to 22). Note that the header space is independent of the path space, so the possible search states in this BFS only depend on the total number of nodes in the lattice (i.e., no state explosion happens even in large-scale networks (§4.6.2)).

Appendix C

Omitted proof and algorithm in chapter 5

C.1 Proof of Algorithm 3

Here, we prove that header sets, which Algorithm 3 computes, satisfy three conditions of Definition 5.

Theorem 4. *Algorithm 3 outputs a set of headers that satisfy all three conditions of Definition 5.*

Proof. First, we prove that the header sets satisfy condition (3). The algorithm generates a set of headers, each with the same forwarding path by checking one-hop forwarding paths on every network device and using a unique hash table *path_to_EC[path]*. Thus, packet headers in the same header set are forwarded to the same output ports on arbitrary network devices. Therefore, the header sets satisfy condition (3). Second, we prove that the header sets satisfy condition (2). For each packet header, we have its forwarding path, and the algorithm assigns a unique EC label to the path using *path_to_EC[path]*. The algorithm sees the packet headers that have the same forwarding path as a set of packet headers belonging to an EC labelled by the EC label. This fact implies that if an EC and another EC have different EC labels, the header sets for the ECs are disjoint. Therefore, the header sets satisfy condition (2). Finally, we prove that the header sets satisfy condition (1). The algorithm computes EC labels for all headers generated by forwarding rules and access control rules by the iteration in line 1. Thus, each packet header is at least a member of a header set. This implies that a union of the header sets equals H. Therefore, the header sets satisfy condition (1). □

C.2 Algorithm for verifying end-to-end reachability

Here, we describe an algorithm to verify end-to-end reachability given a data plane update. Algorithm 8 uses a DFS-based search to check whether a source device d reaches a destination device *dst*. Using Algorithm 4, the algorithm stops when it finds a forwarding loop in the search. Note that this algorithm can be also applied to static reachability verification.

Algorithm 8 starts by adding the new route as a directed edge to the EC-labelled graph G (line 22). It then checks whether the route affects existing ECs (line 23). To this end, it traverses the radix tree to get the EC labels since we maintain the ECs using the radix

tree and the hash table (Algorithm 3). If some ECs are affected by the route update, the algorithm traverses the graph G with affected EC labels and their transition labels if it exist (lines 24 to 31).

The function *ReachDFS* is a DFS-based reachability algorithm. It handles header transformations in the traversal by rewriting the EC label with the transition label to a new one using the hash function *Hs* (lines 4 and 5).

Similarly, we can verify other network properties by modifying the condition to terminate the search (line 3). For instance, if we want to check a hop limit property, we change the condition to check if the depth is greater than the hop limit.

Algorithm 8: Verifying end-to-end reachability

Input: a header for a new route *hdr*, a vertex on EC-labelled graph G for the route v, a vertex on EC-labelled graph for the destination *dst*, a radix tree T
Output: Boolean for end-to-end reachability

```
1  Function ReachDFS(v, dst, ecl, trl, depth):
2      if v == dst then
3          return True
4      if Hs[trl][ecl][v] ≠ null then
5          ecl ← Hs[trl][ecl][v].first
6          if Hs[trl][ecl][v].second = Encap then
7              ECLStack.Push(ecl)
8          if Hs[trl][ecl][v].second = Decap then
9              ECLStack.Pop()
10         if Hs[trl][ecl][v].second = Rewrite then
11             ECLStack.Swap(ecl)
12     Nexthops ← GetNexthop(G, v, ecl)
13     for n in Nexthops do
14         if LookCheck(n) then
15             return False
16         else
17             VtoECLStack[n] ← ECLStack
18             return ReachDFS(n, dst, ecl, trl, depth + 1)
19     return False
20 Function ReachCheck(hdr, v, dst, T):
21     result ← True
22     AddNewRoute(G, v)
23     UpdatedECs ← GetUpdatedEC(hdr, T)
24     for uec in UpdatedECs do
25         ECLStack, VtoECLStack ← New
26         VtoECLStackVec ← New
27         ECLStack.Push(uec.ecl)
28         VtoECLStack[v] ← ECLStack
29         if ReachDFS(v, dst, uec.ecl, uec.trl, 0) = False then
30             report reachability_failure
31             result ← False
32     UpdateECs(T, hdr)
```

C.3 Soundness of Algorithm 8

Theorem 5. *For any EC-labelled graph G, vertex v, EC label ecl, transition label trl, located label $ll = \langle v, ecl \rangle$, and step $i \geq 0$, if not $N \otimes \langle i, ll \rangle$ then after calling ReachCheck(hdr, v, dst, T) there will eventually be a call to ReachDFS(w, dst, ecl', trl, i) for some vertex w and EC label ecl' such that $top([\![G]\!]_i ll) \in \langle w, ecl' \rangle$.*

Proof. The proof is by induction on the step i.

Base Case (i=0) By assumption, we have a located label $ll = \langle v, ecl \rangle$. Using the definition of the semantics $[\![G]\!]$ for the step i, we obtain the following equality:

$$[\![G]\!]_i \langle v, ecl \rangle = \varepsilon \cdot \langle v, ecl \rangle. \tag{C.1}$$

Thus, we must prove that there is a call to $ReachDFS(v, dst, ecl, trl, 0)$. First, *ReachCheck(hdr, v, dst, T)* calls $GetUpdatedEC(h, T)$ (line 23). Afterward, we get an EC label ecl and a transition label trl. Thus, we conclude that there is a call to $ReachDFS(v, dst, ecl, trl, 0)$ where $v = w$ and $ecl = ecl'$.

Inductive Case (i>0) We proceed the proof by using the inductive hypothesis for step $i-1$ to prove that the statement holds for step i. We list out our assumptions from the proof statement as well as the induction hypothesis below:

- not $G \otimes \langle i-1, ll \rangle$
- not $G \otimes \langle i, ll \rangle$
- $top([\![G]\!]_{i-1} ll) = \langle u, ecl'' \rangle$
- $top([\![G]\!]_i ll) = \langle n, ecl''' \rangle$
- there was a call to $ReachDFS(u, dst, ecl'', trl, i-1)$ for some u at step $i-1$.

Given these assumptions, we must prove that there is a call to $ReachDFS(n, dst, ecl''', trl, i)$ for some vertex n and EC label ecl'''.

Case 1: $Hs[trl][ecl''][u] = null$:

By our assumption, the results of the conditional branches in ReachDFS (line 2 for destination check and line 5 for ECL rewriting) are false. By our inductive hypothesis, we get some vertices as nexthops that contains n by calling $GetNexthop(G, u, ecl'')$ (line 12). Finally, we call ReachDFS using these vertices (line 18). Since we know there is no EC label rewriting by our assumption, we compute $ecl'' = ecl'''$. Therefore, we conclude that there is a call to $ReachDFS(n, dst, ecl''', trl, i)$ for some vertex n and EC label ecl'''.

Case 2: $Hs[trl][ecl''][u] \neq null$:

By our assumption, the result of the conditional branch in ReachDFS (line 5 for ECL rewriting) is true, and ecl'' is rewritten to some EC label ecl'''. In addition, ECLStack is modified on the basis of the header transformation type (lines 6, 8, and 10). By the way

to create the hash function Hs in Algorithm 5 and Algorithm 6, we get some vertices as nexthops that contains n using the EC label ecl''' by calling $GetNexthop(G, u, ecl''')$ (line 12). Finally, we call the function ReachDFS using these vertices and EC-label ecl''' (line 18). Therefore, we conclude that there is a call to $ReachDFS(n, dst, ecl''', trl, i)$ for some vertex n and EC label ecl'''.

□

C.4 Two loop conditions

Theorem 6. *Given an EC-labelled graph G, a located label ll induces a loop if and only if either of the two conditions is satisfied such that:*

1. *traversing a vertex using the same EC-label stack that has been used in the previous traversal for the vertex (loop condition (1))*
2. *traversing a vertex while stacking the same (stack of) EC labels that have been used in the previous traversal for the vertex (loop condition (2))*

Proof. We prove the necessity and the sufficiency step by step below. Recall that we store the current EC label stack in traversing every vertex on the EC-labelled graph G as with Algorithm 4.

Sufficient (⇐) We first prove that loop condition (1) induces a loop following Definition 8. By assumption, we traverse a vertex using the same EC-label stack that has been used in the previous traversal for the vertex. Since packet forwarding behavior is deterministic, the re-traversal procedure is repeated using the same EC-label stack periodically. The intermediate vertices are also traversed using the same EC-label stack that has been used in the previous traversal for the vertices. Therefore, loop condition (1) induces the loop following Definition 8.

We prove that loop condition (2) induces the loop. By assumption, we know that the EC-label stack for the vertex will always be stacked with the same (stack of) EC labels in the traversal. Due to the deterministic packet forwarding, the traversal will be repeated, which induces the loop.

Necessary (⇒) By assumption, a located label *ll* induces a forwarding loop following Definition 8. The definition denotes that we periodically traverse vertices in the loop once we traverse the vertex at its start point. To prove the necessity, we need to prove that the assumption implies either loop condition (1) or (2). Thus, we prove that the assumption implies loop condition (1). By assumption, we know that we periodically traverse vertices in the loop. Since packet forwarding behavior is deterministic and header space is finite, we eventually traverse some vertex using the same EC-label stack that has been used in the previous traversal for the vertex. Therefore, the assumption implies the condition (1).

□

Publications

List of the publications related to this PhD dissertation.

Journal papers (peer-reviewed)

1. **R. Shiiba**, S.Kobayashi, O.Akashi, H.Shirokura, K.Fukuda, "Verifying Network-level Properties for Large-scale Networks with Header Transformations in Realtime". Journal of Information Processing, vol.33, pp.41-54, IPSJ, Feb, 2025. DOI: 10.2197/ipsjip.33.41.

International conference papers (peer-reviewed)

1. **R. Shiiba**, S.Kobayashi, O.Akashi, K.Fukuda, "Refining Specifications for Configuration Repair with Side Effect Diagnosis". Proc. Workshop on Formal Methods Aided Network Operation (FMANO'25), pp.43-48, ACM, Coimbra, Portugal, Sep, 2025. DOI: 10.1145/3750022.3750464

International conference papers (under submission)

1. **R. Shiiba**, S.Kobayashi, O.Akashi, K.Fukuda, "Iterative Specification Refinement for Side Effect Free Configuration Repair".

2. **R. Shiiba**, R. Nakamura, Y. Kuga , S.Kobayashi, O.Akashi, K.Fukuda, "Symbolic Extraction of Packet Forwarding Behaviors to Identify Causes and Impacts of Network Failures".

References

[1] *DIGITAL 2025 GLOBAL OVERVIEW REPORT*. 2025. URL: https://wearesocial.com/wp-content/uploads/2025/02/GDR-2025-v2.pdf.

[2] *Cloudflare 2024 Year in Review*. 2024. URL: https://blog.cloudflare.com/radar-2024-year-in-review/.

[3] *The Facebook Data Center FAQ*. 2010. URL: https://www.datacenterknowledge.com/data-center-faqs/facebook-data-center-faq.

[4] N. Feamster and H. Balakrishnan. "Detecting BGP Configuration Faults with Static Analysis." In: *Proc. USENIX NSDI'05*. Boston, MA, 2005, pp. 43–56.

[5] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure." In: *Proc. ACM SIGCOMM'16*. New York, NY, USA, 2016, pp. 58–72. ISBN: 9781450341936.

[6] Q. Zhang, G. Zhao, H. Xu, Z. Yu, L. Xie, Y. Zhao, C. Qiao, Y. Xiong, and L. Huang. "Zeta: A Scalable and Robust East-West Communication Framework in Large-Scale Clouds." In: *USENIX NSDI'22*. Renton, WA, Apr. 2022, pp. 1231–1248. ISBN: 978-1-939133-27-4.

[7] M. Dalton et al. "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization." In: *USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 373–387. ISBN: 978-1-939133-01-4.

[8] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. "CrystalNet: Faithfully Emulating Large Production Networks." In: *Proc. ACM SOSP '17*. Shanghai, China, 2017, pp. 599–613. ISBN: 9781450350853.

[9] F. Ye et al. "Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN." In: *Proc. ACM SIGCOMM'20*. Virtual, 2020, pp. 599–614. ISBN: 9781450379557.

[10] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng. "Running BGP in Data Centers at Scale." In: *Proc. USENIX NSDI'21*. Virtual, Apr. 2021, pp. 65–81. ISBN: 978-1-939133-21-2.

[11] A. Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network." In: *Proc. ACM SIGCOMM'15*. London, UK, 2015, pp. 183–197. ISBN: 9781450335423.

[12] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé. "Semi-Oblivious Traffic Engineering: The Road Not Taken." In: *Proc. USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 157–170. ISBN: 978-1-939133-01-4.

[13] N. Kodirov, S. Bayless, F. Ruffy, I. Beschastnikh, H. H. Hoos, and A. J. Hu. "VNF Chain Allocation and Management at Data Center Scale." In: *Proc. ANCS'18*. Ithaca, New York, 2018, pp. 125–140. ISBN: 9781450359023.

[14] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache. "Dynamic Pricing and Traffic Engineering for Timely Inter-Datacenter Transfers." In: *Proc. ACM SIGCOMM'16*. Florianopolis, Brazil, 2016, pp. 73–86. ISBN: 9781450341936.

[15] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr. "Metron: NFV Service Chains at the True Speed of the Underlying Hardware." In: *Proc. USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 171–186. ISBN: 978-1-939133-01-4.

[16] J. Bogle, N. Bhatia, M. Ghobadi, I. Menache, N. Bjørner, A. Valadarsky, and M. Schapira. "TEAVAR: Striking the Right Utilization-Availability Balance in WAN Traffic Engineering." In: *Proc. ACM SIGCOMM'19*. Beijing, China, 2019, pp. 29–43. ISBN: 9781450359566.

[17] *Anonymized for review.*

[18] S. Jain et al. "B4: Experience with a Globally-Deployed Software Defined Wan." In: *Proc. ACM SIGCOMM'13*. Hong Kong, China, 2013, pp. 3–14. ISBN: 9781450320566.

[19] A. H. Richard Lawler. *Facebook is back online after a massive outage that also took down Instagram, WhatsApp, Messenger, and Oculus*. 2021. URL: https://www.theverge.com/2021/10/4/22708989/instagram-facebook-outage-messenger-whatsapp-error.

[20] A. Toonk. *BGP leak causing internet outages in Japan and beyond*. 2017. URL: https://bgpmon.net/bgp-leak-causing-internet-outages-in-japan-and-beyond/.

[21] M. Robuck. *Due to a router misconfiguration in Atlanta, Cloudflare suffers a network outage on part of its network on Friday*. 2020. URL: https://www.fiercetelecom.com/telecom/due-to-a-router-misconfiguration-cloudflare-suffers-short-outage-friday.

[22] H. H. Liu, X. Wu, W. Zhou, W. Chen, T. Wang, H. Xu, L. Zhou, Q. Ma, and M. Zhang. "Automatic Life Cycle Management of Network Configurations." In: *Proc. ACM SelfDN'18*. Budapest, Hungary, 2018, pp. 29–35. ISBN: 9781450359146.

[23] Z. Whittaker. *T-mobile hit by phone calling, text message outage*. 2023. URL: https://techcrunch.com/2020/06/15/t-mobile-calling-outage/.

[24] *Microsoft Outage Analysis: November 25, 2024*. 2024. URL: https://www.thousandeyes.com/blog/microsoft-outage-analysis-november-25-2024y.

[25] Y. Yuan et al. "New Evolution of Hoyan: Enhancing Scalability, Usability, and Accuracy for Alibaba's Global WAN Verification." In: *Proc. ACM SIGCOMM'25*. São Francisco Convent, Coimbra, Portugal, 2025, pp. 809–825. ISBN: 9798400715242.

[26] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks." In: *Proc. USENIX NSDI'14*. Seattle, WA, 2014, pp. 87–99. ISBN: 978-1-931971-09-6.

[27] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng. "Robotron: Top-down Network Management at Facebook Scale." In: *Proc. ACM SIGCOMM'16*. Florianopolis, Brazil, 2016, pp. 426–439.

[28] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. "NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion." In: *Proc. USENIX NSDI*. 2018, pp. 579–594. ISBN: 978-1-939133-01-4.

[29] R. Yang, H. Shao, G. Han, Z. Wang, X. Fang, L. You, Q. Xiang, L. Kong, R. Zhou, and J. Shu. *Diagnosing and Repairing Distributed Routing Configurations Using Selective Symbolic Simulation*. 2024. arXiv: 2409.20306.

[30] X. Liu, P. Zhang, A. Abhashkumar, J. Chen, and W. Jiang. "Automatic Configuration Repair." In: *Proc. ACM HotNets*. 2024, pp. 213–220. ISBN: 9798400712722.

[31] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. "AED: incrementally synthesizing policy-compliant and manageable configurations." In: *Proc. ACM CoNEXT*. 2020, pp. 482–495. ISBN: 9781450379489.

[32] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. "A General Approach to Network Configuration Analysis." In: *Proc. USENIX NSDI'15*. Oakland, CA, May 2015, pp. 469–483. ISBN: 978-1-931971-218.

[33] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein. "Lessons from the evolution of the Batfish configuration analysis tool." In: *Proc. ACM SIGCOMM'23*. , New York, NY, USA, 2023, pp. 122–135. ISBN: 9798400702365.

[34] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. "A General Approach to Network Configuration Verification." In: *Proc. ACM SIGCOMM '17*. Los Angeles, CA, USA, 2017, pp. 155–168. ISBN: 9781450346535.

[35] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. "Fast Control Plane Analysis Using an Abstract Representation." In: *Proc. ACM SIGCOMM'15*. London, UK, 2016, pp. 300–313. ISBN: 9781450341936.

[36] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li. "Differential Network Analysis." In: *Proc. USENIX NSDI 22*. Renton, WA, Apr. 2022, pp. 601–615. ISBN: 978-1-939133-27-4.

[37] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. "Tiramisu: Fast Multilayer Network Verification." In: *Proc. USENIX NSDI'20*. Santa Clara, CA, Feb. 2020, pp. 201–219. ISBN: 978-1-939133-13-7.

[38] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. "Plankton: Scalable network configuration verification through model checking." In: *Proc. USENIX NSDI'20*. Santa Clara, CA, Feb. 2020, pp. 953–967. ISBN: 978-1-939133-13-7.

[39] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker. "Modular Control Plane Verification via Temporal Invariants." *Proc. ACM Program. Lang.*, (), 2023.

[40] P. Zhang, D. Wang, and A. Gember-Jacobson. "Symbolic Router Execution." In: *Proc. ACM SIGCOMM'22*. New York, NY, USA, 2022, pp. 336–349. ISBN: 9781450394208.

[41] H. Yang and S. S. Lam. "Real-Time Verification of Network Properties Using Atomic Predicates." *IEEE/ACM Transactions on Networking*, (), 2016, pp. 887–900. ISSN: 1063-6692.

[42] H. Yang and S. S. Lam. "Scalable Verification of Networks With Packet Transformers Using Atomic Predicates." *IEEE/ACM Transactions on Networking*, **25**(5), 2017, pp. 2900–2915.

[43] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. "APKeep: Realtime Verification for Real Networks." In: *Proc. USENIX NSDI'20*. Santa Clara, CA, Feb. 2020, pp. 241–255. ISBN: 978-1-939133-13-7.

[44] R. Beckett and A. Gupta. "Katra: Realtime Verification for Multilayer Networks." In: *Proc. USENIX NSDI'20*. Renton, WA, 2022, pp. 617–634. ISBN: 978-1-939133-27-4.

[45] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time." In: *Proc. USENIX NSDI'13*. Lombard, IL, Apr. 2013, pp. 15–27. ISBN: 978-1-931971-00-3.

[46] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. "ddNF: An Efficient Data Structure for Header Spaces." In: *Proc. HVC'16*. Haifa, Israel, 2016, pp. 49–64.

[47] A. Horn, A. Kheradmand, and M. Prasad. "Delta-net: Real-time Network Verification Using Atoms." In: *Proc. USENIX NSDI'17*. Boston, MA, Mar. 2017, pp. 735–749. ISBN: 978-1-931971-37-9.

[48] D. Guo, S. Chen, K. Gao, Q. Xiang, Y. Zhang, and Y. R. Yang. "Flash: Fast, Consistent Data Plane Verification for Large-Scale Network Settings." In: *Proc. ACM SIGCOMM'22*. Amsterdam, Netherlands, 2022, pp. 314–335. ISBN: 9781450394208.

[49] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. "On static reachability analysis of IP networks." In: *Proc. IEEE INFOCOM'05*. Vol. 3. Miami, FL, 2005, pp. 2170–2183.

[50] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. "Real Time Network Policy Checking Using Header Space Analysis." In: *Proc. USENIX NSDI'13*. Lombard, IL, Apr. 2013, pp. 99–111. ISBN: 978-1-931971-00-3.

[51] X. Liu, P. Zhang, H. Li, and W. Sun. "Modular Data Plane Verification for Compositional Networks." *Proc. ACM Netw.*, **1**(), 2023.

[52] Z. Li, P. Zhang, Y. Zhang, and H. Yang. "NDD: A Decision Diagram for Network Verification." In: *Proc. USENIX NSDI'25*. Philadelphia, PA, Apr. 2025, pp. 237–258. ISBN: 978-1-939133-46-5.

[53] J. Backes et al. "Reachability analysis for AWS-based networks." In: *Proc. CAV 2019*. New York, NY, 2019, pp. 231–241.

[54] R. Beckett and R. Mahajan. "Putting network verification to good use." In: *Proc. ACM HotNet's 19*. Princeton, NJ, USA, 2019, pp. 77–84. ISBN: 9781450370202.

[55] X. Xu, Y. Yuan, Z. Kincaid, A. Krishnamurthy, R. Mahajan, D. Walker, and E. Zhai. "Relational Network Verification." In: *Proc. SIGCOMM'24*. 2024.

[56] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. "Config2Spec: Mining Network Specifications from Network Configurations." In: *Proc. USENIX NSDI'20*. Santa Clara, CA, Feb. 2020, pp. 969–984. ISBN: 978-1-939133-13-7.

[57] N. Kang, P. Zhang, H. Li, S. Wen, C. Ji, and Y. Yang. "Network Specification Mining with High Fidelity and Scalability." In: *Proc. IEEE ICNP'23*. Los Alamitos, CA, USA, 2023, pp. 1–11.

[58] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static Checking for Networks." In: *Proc. USENIX NSDI'12*. San Jose, CA, 2012, pp. 113–126. ISBN: 978-931971-92-8.

[59] M. T. Arashloo, R. Beckett, and R. Agarwal. "Formal Methods for Network Performance Analysis." In: *Proc. USENIX NSDI'23*. Apr. 2023, pp. 645–661. ISBN: 978-1-939133-33-5.

[60] A. Seyhani, J. Zhao, A. Gupta, D. Walker, and M. T. Arashloo. "Buffy: A Formal Language-Based Framework for Network Performance Analysis." In: *Proc. ACM HotNets'24*. Irvine, CA, USA, 2024, pp. 95–102. ISBN: 9798400712722.

[61] F. Gong, D. Raghunathan, A. Gupta, and M. Apostolaki. "Towards Integrating Formal Methods into ML-Based Systems for Networking." In: *Proc. ACM HotNets'23*. Cambridge, MA, USA, 2023, pp. 48–55. ISBN: 9798400704154.

[62] C. Guo et al. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis." In: *Proc. ACM SIGCOMM'15*. London, United Kingdom, 2015, pp. 139–152. ISBN: 9781450335423.

[63] P. L. A. Adams and H. Zeng. *"NetNORAD: Troubleshooting Networks via End-to-End Probing,* 2016. URL: https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[64] K. Liu et al. "R-Pingmesh: A Service-Aware RoCE Network Monitoring and Diagnostic System." In: *Proc. ACM SIGCOMM'24*. Sydney, NSW, Australia, 2024, pp. 554–567. ISBN: 9798400706141.

[65] cloudprober. *Cloudprober: Reliable System Monitoring, Simplified!* 2024. URL: https://github.com/cloudprober/cloudprober.

[66] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. "NetBouncer: Active Device and Link Failure Localization in Data Center Networks." In: *Proc. USENIX NSDI'19*. Boston, MA, Feb. 2019, pp. 599–614. ISBN: 978-1-931971-49-2.

[67] R. Ding, X. Liu, S. Yang, Q. Huang, B. Xie, R. Sun, Z. Zhang, and B. Cui. "RD-Probe: Scalable Monitoring With Sufficient Coverage In Complex Datacenter Networks." In: *Proc. ACM SIGCOMM'24*. Sydney, NSW, Australia, 2024, pp. 258–273. ISBN: 9798400706141.

[68] K. Liu et al. "Hostmesh: Monitor and Diagnose Networks in Rail-optimized RoCE Clusters." In: *Proc. ACM APNet'24*. Sydney, Australia, 2024, pp. 122–128. ISBN: 9798400717581.

[69] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. (Liu, J. Padhye, B. T. Loo, and G. Outhred. "007: Democratically Finding the Cause of Packet Drops." In: *Proc. USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 419–435. ISBN: 978-1-939133-01-4.

[70] C. Wang et al. "Towards LLM-Based Failure Localization in Production-Scale Networks." In: *Proc. ACM SIGCOMM'25*. São Francisco Convent, Coimbra, Portugal, 2025, pp. 496–511. ISBN: 9798400715242.

[71] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. "Trumpet: Timely and Precise Triggers in Data Centers." In: *Proc. ACM SIGCOMM'16*. Florianopolis, Brazil, 2016, pp. 129–143. ISBN: 9781450341936.

[72] Y. Zhu et al. "Packet-Level Telemetry in Large Datacenter Networks." *Proc. ACM SIGCOMM'15*, **45**(4), Aug. 2015, pp. 479–491. ISSN: 0146-4833.

[73] Z. Wang et al. "Diagnosing Application-network Anomalies for Millions of IPs in Production Clouds." In: *Proc. USENIX ATC'24*. Santa Clara, CA, July 2024, pp. 885–899. ISBN: 978-1-939133-41-0.

[74] B. Quoitin and S. Uhlig. "Modeling the routing of an autonomous system with C-BGP." *IEEE Network*, **19**(6), 2005, pp. 12–19.

[75] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. "Control Plane Compression." In: *Proc. ACM SIGCOMM'18*. Budapest, Hungary, 2018, pp. 476–489. ISBN: 9781450355674.

[76] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker. *Kirigami, the Verifiable Art of Network Cutting*. 2022.

[77] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker. *Modular Control Plane Verification via Temporal Invariants*. 2022.

[78] A. Tang, R. Beckett, K. Jayaraman, T. Millstein, and G. Varghese. *LIGHTYEAR: Using Modularity to Scale BGP Control Plane Verification*. 2022.

[79] N. Lopes and A. Rybalchenko. "Fast BGP Simulation of Large Datacenters." In: *VMCAI: Verification, Model Checking, and Abstract Interpretation*. Jan. 2019.

[80] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. "Abstract Interpretation of Distributed Network Control Planes." *Proc. ACM Program. Lang.*, (), 2019.

[81] D. Wang, P. Zhang, W. Sun, W. Li, X. Feng, H. Li, J. Chen, W. Jiang, and Y. Tang. "S2: A Distributed Configuration Verifier for Hyper-Scale Networks." In: *Proc. ACM SIGCOMM'25*. São Francisco Convent, Coimbra, Portugal, 2025, pp. 796–808. ISBN: 9798400715242.

[82] L. De Moura and N. Bjørner. "Z3: An Efficient SMT Solver." In: *Proc. TACAS'08/ETAPS'08*. Berlin, Heidelberg, 2008, pp. 337–340. ISBN: 3540787992.

[83] T. Griffin, F. Shepherd, and G. Wilfong. "The stable paths problem and interdomain routing." *IEEE/ACM Transactions on Networking*, (), 2002, pp. 232–243.

[84] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. "Debugging the Data Plane with Anteater." In: *Proc. ACM SIGCOMM'11*. Toronto, Canada, 2011, pp. 290–301. ISBN: 9781450307970.

[85] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. "Checking Beliefs in Dynamic Networks." In: *Proc. USENIX NSDI'15*. Oakland, CA, May 2015, pp. 499–512. ISBN: 978-1-931971-218.

[86] K. Jayaraman et al. "Validating Datacenters at Scale." In: *Proc. ACM SIGCOMM'19*. Beijing, China, 2019, pp. 200–213. ISBN: 9781450359566.

[87] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. "SymNet: Scalable Symbolic Execution for Modern Networks." In: *Proc. ACM SIGCOMM'16*. Florianopolis, Brazil, 2016, pp. 314–327. ISBN: 9781450341936.

[88] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. "Verifying Reachability in Networks with Mutable Datapaths." In: *Proc. USENIX NSDI'17*. Boston, MA, Mar. 2017, pp. 699–718. ISBN: 978-1-931971-37-9.

[89] A. Horn, A. Kheradmand, and M. R. Prasad. "A Precise and Expressive Lattice-theoretical Framework for Efficient Network Verification." In: *Proc. IEEE ICNP'19*. Chicago, IL, 2019, pp. 1–12.

[90] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen. "P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures." In: *Proc. ACM CoNEXT'18*. Heraklion, Greece, 2018, pp. 217–227. ISBN: 9781450360807.

[91] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba. "AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks." In: *Proc. ACM CoNEXT'20*. 2020, pp. 474–481. ISBN: 9781450379489.

[92] Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: vol. C-35. 8. 1986, pp. 677–691.

[93] Q. Xiang, C. Huang, R. Wen, Y. Wang, X. Fan, Z. Liu, L. Kong, D. Duan, F. Le, and W. Sun. "Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification." In: *Proc. ACM SIGCOMM'23*. New York, NY, USA, 2023, pp. 152–166. ISBN: 9798400702365.

[94] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa. "NetConfEval: Can LLMs Facilitate Network Configuration?" *Proc. ACM Netw.*, **2**(), June 2024.

[95] J. Liu, L. Chen, D. Li, and Y. Miao. "CEGS: Configuration Example Generalizing Synthesizer." In: *Proc. USENIX NSDI'25*. Philadelphia, PA, Apr. 2025, pp. 1327–1347. ISBN: 978-1-939133-46-5.

[96] X. Liu, P. Zhang, A. Abhashkumar, J. Chen, and W. Jiang. "Automatic Configuration Repair." In: *Proc. ACM HotNets'24*. New York, NY, USA, 2024, pp. 213–220. ISBN: 9798400712722.

[97] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu. "Automatically Repairing Network Control Planes Using an Abstract Representation." In: *Proc. ACM SOSP*. New York, NY, USA, 2017, pp. 359–373. ISBN: 9781450350853.

[98] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. "AED: incrementally synthesizing policy-compliant and manageable configurations." In: *Proc. ACM CoNEXT'20*. New York, NY, USA, 2020, pp. 482–495. ISBN: 9781450379489.

[99] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. "NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion." In: *Proc. USENIX NSDI'18*. Apr. 2018, pp. 579–594. ISBN: 978-1-939133-01-4.

[100] R. Mondal, A. Tang, R. Beckett, T. Millstein, and G. Varghese. "What do LLMs need to Synthesize Correct Router Configurations?" In: *Proc. ACM HotNets'23*. Cambridge, MA, USA, 2023, pp. 189–195. ISBN: 9798400704154.

[101] S. Ramanathan, Y. Zhang, M. Gawish, Y. Mundada, Z. Wang, S. Yun, E. Lippert, W. Taha, M. Yu, and J. Mirkovic. "Practical Intent-driven Routing Configuration Synthesis." In: *Proc. USENIX NSDI'23)*. Boston, MA, Apr. 2023, pp. 629–644. ISBN: 978-1-939133-33-5.

[102] H. Hojjat, P. Rümmer, J. McClurg, P. Černý, and N. Foster. "Optimizing horn solvers for network repair." In: *Proc. ACM FMCAD'16*. Mountain View, California, 2016, pp. 73–80. ISBN: 9780983567868.

[103] R. Mondal, N. Bjorner, T. Millstein, A. Tang, and G. Varghese. *LLM-Based Config Synthesis requires Disambiguation*. 2025. arXiv: 2507.12443 [cs.NI]. URL: https://arxiv.org/abs/2507.12443.

[104] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. "Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations." In: *Proc. ACM SIGCOMM'16*. Florianopolis, Brazil, 2016, pp. 328–341. ISBN: 9781450341936.

[105] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. "Network configuration synthesis with abstract topologies." *Proc. ACM PLDI'17*, **52**(6), June 2017, pp. 437–451. ISSN: 0362-1340.

[106] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. "Network-Wide Configuration Synthesis." In: *Proc. Springer CAV'17*. Ed. by R. Majumdar and V. Kunčak. 2017, pp. 261–281. ISBN: 978-3-319-63390-9.

[107] P. Agarwal and A. P. Agrawal. "Fault-localization techniques for software systems: a literature review." *SIGSOFT Softw. Eng. Notes*, **39**(5), Sept. 2014, pp. 1–8. ISSN: 0163-5948.

[108] J. A. Jones and M. J. Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." In: *Proc. IEEE/ACM ASE'05*. ASE '05. Long Beach, CA, USA, 2005, pp. 273–282. ISBN: 1581139934.

[109] R. Mondal, A. Tang, R. Beckett, T. Millstein, and G. Varghese. "What do LLMs need to Synthesize Correct Router Configurations?" In: *Proc. ACM HotNets'25*. Cambridge, MA, USA, 2023, pp. 189–195. ISBN: 9798400704154.

[110] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. "Management Plane Analytics." In: *Proc. ACM IMC'15*. New York, NY, USA, 2015, pp. 395–408. ISBN: 9781450338486.

[111] S. Vissicchio, L. Vanbever, C. Pelsser, L. Cittadini, P. Francois, and O. Bonaventure. "Improving Network Agility With Seamless BGP Reconfigurations." *IEEE/ACM Transactions on Networking*, **21**(3), 2013, pp. 990–1002.

[112] A. Mahimkar, C. E. de Andrade, R. Sinha, and G. Rana. "A composition framework for change management." In: *Proc. 2021 ACM SIGCOMM'21*. Virtual Event, USA, 2021, pp. 788–806. ISBN: 9781450383837.

[113] A. H. Richard Lawler. *Facebook is back online after a massive outage that also took down Instagram, WhatsApp, Messenger, and Oculus*. 2021. URL: https://www.theverge.com/2021/10/4/22708989/instagram-facebook-outage-messenger-whatsapp-error.

[114] M. Robuck. *Due to a router misconfiguration in Atlanta, Cloudflare suffers a network outage on part of its network on Friday*. 2020. URL: https://www.fiercetelecom.com/telecom/due-to-a-router-misconfiguration-cloudflare-suffers-short-outage-friday.

[115] T. Benson, A. Akella, and D. Maltz. "Unraveling the complexity of network management." In: Proc. USENIX NSDI'09, 2009, pp. 335–348.

[116] C. Wang, M. Scazzariello, A. Farshin, D. Kostic, and M. Chiesa. *Making Network Configuration Human Friendly*. 2023. arXiv: 2309.06342 [cs.NI]. URL: https://arxiv.org/abs/2309.06342.

[117] A. Kheradmand. "Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns." In: *Proc. ACM SOSR '20*. New York, NY, USA, 2020, pp. 27–33. ISBN: 9781450371018.

[118] N. Kang, P. Zhang, H. Li, S. Wen, C. Ji, and Y. Yang. "Network Specification Mining With High Fidelity, Scalability, and Readability." *IEEE Transactions on Networking*, (), 2025, pp. 1–16.

[119] T. Benson, A. Akella, and D. A. Maltz. "Mining policies from enterprise network configuration." In: *Proc. ACM IMC'09*. Chicago, Illinois, USA, 2009, pp. 136–142. ISBN: 9781605587714.

[120] A. Nazari, Y. Zhang, M. Raghothaman, and H. Chen. "Localized Explanations for Automatically Synthesized Network Configurations." In: *Proc. ACM HoteNets'24*. Irvine, CA, USA, 2024, pp. 52–59. ISBN: 9798400712722.

[121] R. Shiiba, S. Kobayashi, O. Akashi, and K. Fukuda. "Refining Specifications for Configuration Repair with Side Effect Diagnosis." In: *Proc. ACM FMANO'25*. Coimbra, Portugal, 2025, pp. 43–48. ISBN: 9798400721038.

[122] R. Yang, X. Fang, L. You, Q. Xiang, H. Shao, g. han gao, Z. Wang, Z. Zhang, J. Shu, and L. Kong. "Diagnosing Distributed Routing Configurations Using Sequential Program Analysis." In: *Proc. ACM APNet'23*. New York, NY, USA, 2023, pp. 34–40. ISBN: 9798400707827.

[123] T. Schneider, R. Birkner, and L. Vanbever. "Snowcap: synthesizing network-wide configuration updates." In: *Proc. ACM SIGCOMM'21*. Virtual Event, USA, 2021, pp. 33–49. ISBN: 9781450383837.

[124] T. Schneider, J. M´egret, and L. Vanbever. "Guided Exploration of Control Plane Routing States." In: *Proc. IEEE ICNP'25*. 2025, pp. 1–11.

[125] NSG-ETHZ. *NetComplete*. 2018. URL: https://github.com/nsg-ethz/synet-plus.

[126] R. H. Research. *NetBuddy*. 2024. URL: https://github.com/RedHatResearch/conext24-NetConfEval.

[127] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. "The Internet Topology Zoo." *IEEE Journal on Selected Areas in Communications*, **29**(9), 2011, pp. 1765–1775.

[128] NSG-ETHZ. *Config2Spec*. 2020. URL: https://github.com/nsg-ethz/config2spec.

[129] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. "Merlin: A Language for Provisioning Network Resources." In: *Proc. ACM CoNEXT'14*. Sydney, Australia, 2014, pp. 213–226. ISBN: 9781450332798.

[130] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao. "Hey, Lumi! Using Natural Language for Intent-Based Network Management." In: *Proc. USENIX ATC'21*. July 2021, pp. 625–639. ISBN: 978-1-939133-23-6.

[131] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. "Supporting Diverse Dynamic Intent-based Policies using Janus." In: *Proc. ACM CoNEXT'17*. Incheon, Republic of Korea, 2017, pp. 296–309. ISBN: 9781450354226.

[132] R. Han, J. Wang, H. Sun, Z. Jiang, Q. Qi, Z. Zhuang, Y. Zhang, and J. Liao. "Network CoPilot: Intent-Driven Network Configuration Updating for Service Guarantee." In: *Proc. IEEE INFOCOM'25*. 2025, pp. 1–10.

[133] T. Schneider, R. Schmid, and L. Vanbever. "On the Complexity of Network-Wide Configuration Synthesis." In: *Proc. IEEE ICNP'22*. 2022, pp. 1–11.

[134] Z. Li, Y. Cao, X. Xu, J. Jiang, X. Liu, Y. S. Teo, S.-W. Lin, and Y. Liu. "LLMs for Relational Reasoning: How Far are We?" In: *Proc. ACM LLM4Code '24*. Lisbon, Portugal, 2024, pp. 119–126. ISBN: 9798400705793.

[135] M. Anwar and M. Caesar. "Understanding Misunderstandings: Evaluating LLMs on Networking Questions." *ACM SIGCOMM Comput. Commun. Rev.*, **54**(4), Feb. 2025, pp. 14–24. ISSN: 0146-4833.

[136] J. R. Mendoza and R. Ocampo. "PeeringLLM-Bench: Evaluating LLMs for BGP Configuration Tasks." In: *Proc, ACM AINTEC'25*. Taguig, Philippines, 2025, pp. 78–86. ISBN: 9798400718465.

[137] B. Tian et al. "Safely and automatically updating in-network ACL configurations with intent language." In: *Proc. ACM SIGCOMM'19*. Beijing, China, 2019, pp. 214–226. ISBN: 9781450359566.

[138] K. Subramanian, L. D'Antoni, and A. Akella. "Synthesis of Fault-Tolerant Distributed Router Configurations." *Proc. ACM Meas. Anal. Comput. Syst.*, **2**(1), Apr. 2018.

[139] L. Beurer-Kellner, M. Vechev, L. Vanbever, and P. Veličković. "Learning to configure computer networks with neural algorithmic reasoning." In: *Proc. NeurIPS'22*. NIPS '22. New Orleans, LA, USA, 2022. ISBN: 9781713871088.

[140] T. Schneider, R. Schmid, S. Vissicchio, and L. Vanbever. "Taming the transient while reconfiguring BGP." In: *Proc. ACM SIGCOMM'23*. New York, NY, USA, 2023, pp. 77–93. ISBN: 9798400702365.

[141] F. Clad, S. Vissicchio, P. Mérindol, P. Francois, and J.-J. Pansiot. "Computing minimal update sequences for graceful router-wide reconfigurations." *IEEE/ACM Trans. Netw.*, **23**(5), Oct. 2015, pp. 1373–1386. ISSN: 1063-6692.

[142] P. Francois, M. Shand, and O. Bonaventure. "Disruption Free Topology Reconfiguration in OSPF Networks." In: *Proc. IEEE INFOCOM'07*. 2007, pp. 89–97.

[143] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. "Automatic Test Packet Generation." In: *Proc. ACM CoNEXT'12*. Nice, France, 2012, pp. 241–252. ISBN: 9781450317757.

[144] G. V. Hongyi Zeng Peyman Kazemian and N. McKeown. *A survey on network trouble shooting*. http://yuba.stanford.edu/~peyman/docs/atpg-survey.pdf. 2012.

[145] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors." In: *Proc. USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 609–623. ISBN: 978-1-931971-43-0.

[146] Y. Fang and Y. Lu. "Real-Time Verification of Network Properties Based on Header Space." *IEEE Access*, (), 2020, pp. 36789–36806.

[147] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press, 2002.

[148] V. Harsh, T. Meng, K. Agrawal, and P. B. Godfrey. "Flock: Accurate Network Fault Localization at Scale." *Proc. ACM Netw.*, **1**(), 2023.

[149] X. Zuo, Q. Li, J. Xiao, D. Zhao, and J. Yong. "Drift-Bottle: A Lightweight and Distributed Approach to Failure Localization in General Networks." In: *Proc. ACM CoNEXT'22*. New York, NY, USA, 2022, pp. 337–348. ISBN: 9781450395083.

[150] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. "Efficient querying and maintenance of network provenance at internet-scale." In: *Proc.ACM SIGMOD'10*. New York, NY, USA, 2010, pp. 615–626. ISBN: 9781450300322.

[151] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. "The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance." In: *Proc. ACM SIGCOMM'16*. New York, NY, USA, 2016, pp. 115–128. ISBN: 9781450341936.

[152] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. "Diagnosing missing events in distributed systems with negative provenance." In: *Proc. ACM SIGCOMM'14*. New York, NY, USA, 2014, pp. 383–394. ISBN: 9781450328364.

[153] B. Tian et al. "Safely and automatically updating in-network ACL configurations with intent language." In: *Proc. ACM SIGCOMM'19*. New York, NY, USA, 2019, pp. 214–226. ISBN: 9781450359566.

[154] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. S. E. Ng. "Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon." In: *Proc. USENIX NSDI'22*. Renton, WA, 2022, pp. 267–285. ISBN: 978-1-939133-27-4.

[155] D. Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *Proc. USENIX NSDI'18*. Renton, WA, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4.

[156] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. "TIMELY: RTT-based Congestion Control for the Datacenter." In: *Proc. ACM SIGCOMM CCR'15*. New York, NY, USA, 2015, pp. 537–550. ISBN: 9781450335423.

[157] F. Le, G. G. Xie, and H. Zhang. "On route aggregation." In: *Proc. ACM CoNEXT'11*. New York, NY, USA, 2011. ISBN: 9781450310413.

[158] D. G. Kourie, S. Obiedkov, B. W. Watson, and D. van der Merwe. "An incremental algorithm to construct a lattice of set intersections." *Science of Computer Programming*, (), 2009, pp. 128–142. ISSN: 0167-6423.

[159] C. Clos. "A study of non-blocking switching networks." *The Bell System Technical Journal*, **32**(2), 1953, pp. 406–424.

[160] *Anonymized for review.*

[161] sngroup-xmu. *Tulkun*. https://github.com/sngroup-xmu/ddpv-pubilc/tree/main. 2023.

[162] H. Asai. "Palmtrie: A Ternary Key Matching Algorithm for IP Packet Filtering Rules." In: *Proc. ACM CoNEXT'20*. New York, NY, USA, 2020, pp. 323–335. ISBN: 9781450379489.

[163] D. E. Taylor and J. S. Turner. "ClassBench: A Packet Classification Benchmark." *IEEE/ACM Trans. Netw.*, (), 2007, pp. 499–511. ISSN: 1063-6692.

[164] C. E. Perkins. *IP Encapsulation within IP*. RFC 2003. Oct. 1996.

[165] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. "Network Monitoring as a Streaming Analytics Problem." In: *Proc. ACM HotNet'16*. New York, NY, USA, 2016, pp. 106–112. ISBN: 9781450346610.

[166] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. "SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks." In: *Proc. USENIX NSDI'19*. Boston, MA, Feb. 2019, pp. 549–564. ISBN: 978-1-931971-49-2.

[167] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems." In: *Proc. ACM ASPLOS'16*. Atlanta, Georgia, USA, 2016, pp. 517–530. ISBN: 9781450340915.

[168] Q. Xiang, R. Wen, C. Huang, Y. Wang, and F. Le. "Network can check itself: scaling data plane checking via distributed, on-device verification." In: *Proc. ACM HotNet'22*. Austin, Texas, 2022, pp. 85–92. ISBN: 9781450398992.

[169] R. Schmid, T. Schneider, G. Fragkouli, and L. Vanbever. "Predicting Specification Violations During BGP Convergence." In: *Proc. ACM CoNEXT Student Workshop'23*. New York, NY: Association for Computing Machinery, 2023, pp. 25–26. ISBN: 979-8-4007-0452-9.

[170] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar. "NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification." In: *Proc. USENIX NSDI'20*. Santa Clara, CA, 2020, pp. 181–200. ISBN: 978-1-939133-13-7.

[171] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. "Elastic Scaling of Stateful Network Functions." In: *Proc. USENIX NSDI'15*. USA, 2018, pp. 299–312. ISBN: 9781931971430.

[172] H. Shirokura. *Hyperscale distributed NAT system and software engineering*. https://linedevday.linecorp.com/2020/ja/sessions/2076/. LINE DevDay 2020.

[173] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer." In: USA: Proc. USENIX NSDI'16, 2016, pp. 523–535. ISBN: 9781931971294.

[174] P. Patel et al. "Ananta: Cloud Scale Load Balancing." In: *Proc. ACM SIGCOMM'13*. New York, NY, USA, 2013, pp. 207–218. ISBN: 9781450320566.

[175] H. Shirokura. *High Functional Cloud NFV System Design and Implementation at LINE Cloud*. https://www.janog.gr.jp/meeting/janog48/linenfv/. JANOG48. 2021.

[176] Y. Fang and Y. Lu. "Real-Time Verification of Network Properties Based on Header Space." *IEEE Access*, **8**(), 2020, pp. 36789–36806.

[177] Q. Xiang, C. Huang, R. Wen, Y. Wang, X. Fan, Z. Liu, L. Kong, D. Duan, F. Le, and W. Sun. "Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification." In: *Proc. ACM SIGCOMM'23*. New York, NY, USA, 2023, pp. 152–166. ISBN: 9798400702365.

[178] X. Liu, P. Zhang, H. Li, and W. Sun. "Modular Data Plane Verification for Compositional Networks." *Proc. ACM Netw.*, **1**(), 2023.

[179] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li. *Segment Routing over IPv6 (SRv6) Network Programming*. RFC 8986. Feb. 2021. DOI: 10.17487/RFC8986. URL: https://rfc-editor.org/rfc/rfc8986.txt.

[180] F. Project. *FRR*. Version 7.5.1. May 7, 2021. URL: https://frrouting.org/.

[181] S. Labs. *Containerlab*. Version 0.31.1. Aug. 15, 2022. URL: https://containerlab.dev/.

[182] *The Internet2 Observatory Data Collection*. http://www.internet2.edu/observatory/archive/data-collections.html.

[183] *DELTA-NET*. https://github.com/delta-net/datasets.

[184] Q. Xiang, R. Wen, C. Huang, Y. Wang, and F. Le. "Network can check itself: scaling data plane checking via distributed, on-device verification." In: *Proc. ACM HotNets'22*. Austin, Texas, 2022, pp. 85–92. ISBN: 9781450398992.

[185] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev. "Probabilistic Verification of Network Configurations." In: Virtual Event, USA: Proc. ACM SIGCOMM'20, 2020, pp. 750–764. ISBN: 9781450379557.

[186] R. Liu, F. Li, C. Gao, C. Ji, and X. Wang. "NetDiceSyn: Multi-Property Probabilistic Verification of Network Configurations." In: *Proc. IEEE/ACM IWQoS'23*. 2023, pp. 1–10.

[187] D. Raghunathan, M. Apostolaki, and A. Gupta. "A Layered Formal Methods Approach to Answering Queue-related Queries." In: *Proc. USENIX NSDI'25*. Philadelphia, PA, Apr. 2025, pp. 1289–1304. ISBN: 978-1-939133-46-5.

[188] M. T. Arashloo, R. Beckett, and R. Agarwal. "Formal Methods for Network Performance Analysis." In: *Proc. USENIX NSDI'23*. Boston, MA, Apr. 2023, pp. 645–661. ISBN: 978-1-939133-33-5.

[189] P. Zhang, Y. Huang, A. Gember-Jacobson, W. Shi, X. Liu, H. Yang, and Z. Zuo. "Incremental Network Configuration Verification." In: *Proc. ACM HotNets'20*. Virtual Event, USA, 2020, pp. 81–87. ISBN: 9781450381451.