DISS. ETH NO. 27926

IMPROVING NETWORK UNDERSTANDING

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZÜRICH
(Dr. sc. ETH Zürich)

presented by

RÜDIGER JAN BIRKNER
MSc ETH EEIT
ETH Zürich

born on December 10th, 1989
citizen of Zürich ZH, Switzerland and Germany

accepted on the recommendation of

Prof. Dr. Laurent Vanbever (Advisor)
Prof. Dr. Martin Vechev (Co-Advisor)
Prof. Dr. Aaron Gember-Jacobson
Prof. Dr. David Walker

2021

Rüdiger Jan Birkner: *Improving Network Understanding,* © 2021

Diss. ETH No. 27926
TIK-Schriftenreihe-Nr. 193

ABSTRACT

One cannot imagine the world today without the Internet, as it has become an integral part of our daily lives. However, with all the benefits and opportunities it brings also come enormous availability and reliability requirements, which put immense pressure on the operators running the individual networks of the Internet. They need to avoid disruptions at all costs and resolve outages as fast as possible. Unfortunately, this is a highly challenging task due to the sheer complexity of these networks.

This dissertation focuses on assisting network operators in one aspect of their daily work: network understanding. To this end, we built three systems that automate and improve network understanding to allow the network operators to direct their full attention to the mission-critical tasks requiring their expert insights.

First, we developed *Net2Text*, a system which assists network operators in understanding their network's forwarding behavior. Based on the operators' queries, it automatically produces succinct summaries of the raw forwarding state. The key insight behind *Net2Text* is to pose the problem of summarizing the network-wide forwarding state as an optimization problem that aims to balance coverage, by explaining as many paths as possible, and explainability, by maximizing the provided information.

Second, we developed *Config2Spec*, a system which assists network operators in understanding their network's configuration. It automatically mines the network's specification, which consists of all the policies that the configuration enforces. The key insight behind *Config2Spec* is to combine two well-known techniques: data-plane analysis and control-plane verification. This combination allows to prune the large space of candidate policies efficiently and then validate the remaining ones.

And third, we developed *Metha*, a system which helps network operators understand the capabilities of their network validation tools. It finds inaccuracies in the underlying network models using differential testing. The key insight behind *Metha* is to leverage grammar-based fuzzing together with combinatorial testing to ensure thorough coverage of the search space using syntactically- and semantically-valid configurations.

ZUSAMMENFASSUNG

Das Internet ist aus der heutigen Zeit nicht mehr wegzudenken, da es zu einem festen Bestandteil unseres Alltags geworden ist. Mit all den Vorteilen und Möglichkeiten, die es mit sich bringt, gehen auch enorm hohe Anforderung an seine Verfügbarkeit und Zuverlässigkeit einher, die die Betreiber der einzelnen Netzwerke des Internets unter einen immensen Druck setzen. Sie müssen Ausfälle um jeden Preis verhindern und Unterbrüche so schnell wie möglich beheben. Leider ist dies aufgrund der grossen Komplexität dieser Netze eine äusserst schwierige Aufgabe.

Das Hauptaugenmerk dieser Dissertation liegt auf der Unterstützung der Netzwerkbetreiber in einem wichtigen Bereich ihrer täglichen Arbeit: dem Verständnis ihres Netzwerkes. Zu diesem Zweck haben wir drei verschiedene Systeme entwickelt, die das Netzwerk-Verständnis automatisieren und verbessern, damit sich die Betreiber voll und ganz den zentralen Aufgaben widmen können, die ihre gesamte Sachkenntnis erfordern.

Als erstes haben wir *Net2Text* entwickelt, ein System, das Netzwerkbetreiber darin unterstützt, zu verstehen, wie die Daten durch ihr Netzwerk geleitet werden. Basierend auf den Anfragen der Betreiber, erstellt *Net2Text* automatisch kurze und bündige Zusammenfassungen des Netzwerkverhaltens. Der Kerngedanke hinter *Net2Text* ist, das Zusammenfassen des kompletten Netzwerksverhaltens als Optimierungsproblem zu betrachten, welches darauf abzielt, ein Gleichgewicht zwischen dem Umfang und der Detailliertheit der Zusammenfassungen herzustellen.

Als zweites haben wir *Config2Spec* entwickelt, ein System, welches Netzwerkbetreibern dabei hilft, die Konfigurationen ihrer Netzwerke besser zu verstehen. Ausgehend von der Netzwerkkonfiguration, lernt es automatisch alle Regeln und Richtlinien, die die Konfiguration im Netzwerk umsetzt. Die Schlüsselerkenntnis hinter *Config2Spec* besteht aus der Kombination zweier bekannter und erprobter Methoden: der Datenschichtanalyse und der Kontrollschichtverifikation. Diese einzigartige Kombination ermöglicht es, den riesigen Suchraum aller möglichen Netzwerkrichtlinien schnell und effizient einzugrenzen und dann die verbleibenden Richtlinien nach und nach zu überprüfen.

Als drittes haben wir *Metha* entwickelt, ein System, welches Netzwerkbetreiber darin unterstützt, die Stärken und Schwächen ihrer Netzwerkanalyse-Werkzeuge besser zu erkennen und zu verstehen. Mittels differentiellem Testen findet *Metha* automatisch Fehler und Ungenauigkeiten in den Modellen, die diesen Werkzeugen zugrunde liegen. Die wichtigste Erkenntnis hinter *Metha* ist das zufällige Generieren von Konfigurationen basierend auf einer Grammatik zusammen mit kombinatorischem Testen. Dies stellt sicher, dass der ganze Suchraum mittels syntaktisch und semantisch korrekten Konfigurationen gründlich abgedeckt wird.

PUBLICATIONS

This thesis is based on previously published conference proceedings. The list of accepted and submitted publications is presented hereafter.

Net2Text: Query-Guided Summarization of Network Forwarding Behaviors

Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev.
In *USENIX NSDI*, Renton, WA, USA, 2018.

Config2Spec: Mining Network Specifications from Network Configurations

Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev.
In *USENIX NSDI*, Santa Clara, CA, USA, 2020.

Metha: Network Verifiers Need To Be Correct Too!

Rüdiger Birkner*, Tobias Brodmann*, Petar Tsankov, Laurent Vanbever, and Martin Vechev.
In *USENIX NSDI*, Online, 2021.
*These authors contributed equally to this work.

The remaining publications were part of my PhD research, but are not covered in this thesis. The topics of these publications are outside of the scope of the material covered here.

Network Monitoring as a Streaming Analytics Problem

Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger.
In *ACM HotNets*, Atlanta, GA, USA, 2016.

Concise Encoding of Flow Attributes in SDN Switches

Robert MacDavid, Rüdiger Birkner, Ori Rottenstreich, Arpit Gupta, Nick Feamster, and Jennifer Rexford.
In *ACM SOSR*, Santa Clara, CA, USA, 2017.

SDX-Based Flexibility or Internet Correctness? Pick Two!

Rüdiger Birkner, Arpit Gupta, Nick Feamster,
and Laurent Vanbever.
In *ACM SOSR*, Santa Clara, CA, USA, 2017.

Snowcap: Synthesizing Network-Wide Configuration Updates

Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever.
In *ACM SIGCOMM*, Online, 2021.

ACKNOWLEDGMENTS

This dissertation is the result of my research journey over the last couple of years, which would not have been possible without the help and support of many people along the way. I would like to dedicate this section to all these people and mention those that deserve a special thank you.

First and foremost, I would like to thank my two advisors, Professor Laurent Vanbever and Professor Martin Vechev, for their support, trust, and guidance throughout my doctorate. I highly appreciate Laurent's leadership and supervision, which were instrumental to the success of this thesis. Especially, Laurent's emphasis on clear and comprehensive communication helped me develop my presentation and writing skills. I am also grateful to my second advisor, Martin, for introducing me to programming languages and formal methods and providing me with a lot of freedom.

I sincerely thank Aaron Gember-Jacobson and David Walker for devoting the time to be on my dissertation committee and reading this thesis.

I owe a special thanks to Dana Drachsler Cohen, whom I have worked with during large parts of my doctorate. It was always great to discuss and develop ideas together and profit from her research experience.

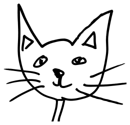
I would also like to thank all my colleagues and friends at the Networked Systems Group (NSG), who have not only supported me in my research, but have also made this time enjoyable: Maria Apostolaki, Thomas Holterbach, Roland Meier, Alexander Dietmüller, Albert Gran Alcoz, Ege Cem Kirci, Roland Schmid, Rui Yang, and Tibor Schneider. I have very fond memories of our board game nights, hikes, lunches, and retreats. Especially, I would like to thank Ahmed El-Hassany, who has been my long-time office mate and mentor. I benefited a lot from his expertise and experience from all his previous gigs. I am also very grateful to Edgar Costa Molero, my "other" office mate, for all the discussions about work and everything else. Edgar has been a great support for me during the last two years of my doctorate. Finally, I would like to thank Tobias Bühler with whom I have worked on many, many things during the doctorate and before. It was always great to know that we have each other's back.

In addition, I would like to acknowledge the members of my "second" group: the Secure, Reliable, Intelligent Systems Lab (SRI). In particular, I would like to thank Benjamin Bichsel, Dimitar Dimitrov, Marc Fischer, Samuel Steffen, and Petar Tsankov for the discussions and collaborations.

I would also like to thank Beat Futterknecht, our administrator, who supported me in all administrative and logistic matters. No request was too complicated: he always managed to help me and work something out.

Over the course of my doctorate, I also got the opportunity to advise and work with several students. I am very grateful to them for their dedication and patience with me. Specifically, I would like to thank Tobias Brodmann, Yu Chen, Philipp Mao, and Tibor Schneider.

Finally, I would like to thank my family and friends who supported me throughout all the years of my doctorate outside of ETH. I highly appreciate that they were always open to talk and distract me from my work and never got angry when I cancelled on them because of some deadline. Special thanks go to my parents and my sisters, Bettina and Dorothea.



Rüdiger Birkner
October 2021

CONTENTS

1 INTRODUCTION

The Internet has become such an integral part of today's world that it is almost impossible to imagine life without it. We rely so many times on the Internet throughout a typical day: It starts with the first glance at the smartphone in the morning to check the emails and the weather forecast. Then, it continues with all the tools at work that enable collaboration with colleagues around the world through file sharing and video conferencing. Finally, the day ends with a dinner ordered through a food delivery platform and a movie streamed to the television.

We take it for granted that the Internet "just works" and one often does not even notice all the places we rely on it. This, however, completely changes when a failure happens and an outage occurs. Suddenly, airplanes cannot take off anymore [1], shopping with cashless payments becomes impossible [2, 3], and emergency numbers are unreachable [4]. It feels like life comes to a standstill and it does not take long until the first customers complain and companies start to lose millions in revenue [5].

Hence, network operators are under immense pressure as they need to ensure high availability and reliability of their networks and have to resolve disruptions as fast as possible during hopefully rare cases of failures.

Operating one of the networks in the Internet is a challenging task. It all starts with the high-level business, reliability, and security objectives, which the network operators aim to enforce in their network through a corresponding configuration. However, such a network is not a monolith that network operators can centrally configure, but rather a distributed system consisting of many routers and switches that all need to be individually configured. Hence, the operators need to translate their network-wide objectives into device-level configurations.

Because of this distributed nature and the, to large parts still manual, configuration process, it comes as no surprise that misconfigurations occur and, unfortunately, more often than one would hope for. A recent study from Alibaba, for example, shows that the majority of their network incidents (56%) in 2016 and 2017 were caused by configuration updates [6].

Over the years, researchers and industry alike have developed several tools to prevent (or minimize) these human-induced mistakes: Configuration synthesizers automatically come up with provably correct configurations for the network and configuration validators allow to check configurations for correct behavior before deploying them in the network.

Unfortunately, the use of these tools is not yet as widespread as one would hope for, especially among the "average" and small networks [7]. The reasons for that are manifold, but two stand out: *(i)* the need for a formal network specification and *(ii)* the accuracy of these tools:

Need for a specification: Both configuration synthesizers and validators need to know the intended behavior of a network Otherwise, they cannot create a corresponding configuration or validate its behavior. To that end, network operators need to provide a formal specification of their objectives as input, which usually does not exist. Traditionally, the specification is defined informally and, in the best case, documented to large parts in network design and architecture documents.

Network model accuracy: All these tools rely on an underlying network model that reproduces the network's behavior. To be helpful, that model does not only need to support the exact combination of routing protocols and configuration features used in the network, but it also has to do that faithfully. This means the model needs to behave exactly like the network, which is extremely difficult to achieve due to the many different vendors, device models, and software versions.

Even with the use of configuration synthesis and validation tools, network outages can still occur. Hence, preventing configuration errors is just one aspect of running a reliable network. Another aspect is resolving these outages and recovering the network rapidly. To this end, network operators aim to pinpoint the root cause as quickly as possible and gather various information about the network's behavior. Unfortunately, this is difficult for two reasons: *(i)* the data is hard to access and *(ii)* there is an overload of low-level data.

Data access: The common debugging tools (e.g., `ping` and `traceroute`) and data sources (e.g., routing tables and traffic statistics) provide rudimentary interfaces for the network operators. In addition, due to the distributed nature of the traditional networks, there is often no central entity collecting the data, instead, the operators have to gather it across the entire network.

Data overload: Networks handle traffic for almost 900 000 destinations [8] and carry traffic at ever-increasing speeds. Network operators can only extract the low-level data, such as device-specific routing tables and traffic statistics, from the network and then have to search through the data to identify the relevant pieces manually.

The underlying problem is the lack of network understanding. In this dissertation, we aim to change that. The overarching goal is to help and assist network operators in managing a safe and reliable network and not to replace them. To this end, we present three systems that each focus on one aspect of network understanding: *(i) Net2Text*, which helps operators better understand a network's current forwarding behavior; *(ii) Config2Spec*, which helps operators better understand a network's configuration and the underlying policies; and *(iii) Metha*, which helps operators better understand the capabilities and accuracy of network validation tools.

Net2Text assists the operator in reasoning about the network-wide forwarding behavior. It automatically gathers the relevant low-level forwarding state and extracts the high-level insights based on the operator's query, provided in natural language. As a result, *Net2Text* produces succinct summaries, which efficiently capture network-wide semantics.

Config2Spec helps the operator understand a network configuration's underlying policies. It automatically extracts the network's specification from its configuration and a failure model. Thus, *Config2Spec* bridges the gap between low-level configurations and high-level network policies.

Metha helps network operators (and developers) better understand the capabilities of their network validation tools in terms of feature coverage and accuracy. It automatically tests these tools and reports any inaccuracies in their network models. Thus, *Metha* helps the operators gain trust in their validation tools and helps developers build more accurate tools.

The rest of the dissertation is organized as follows: First, we provide the necessary background in Chapter 2. In Chapter 3, we present *Net2Text*, an interactive system which assists the network operator in reasoning about the network-wide forwarding state. In Chapter 4, we present *Config2Spec*, a system that automatically mines a network's specification from its configuration and a failure model. In Chapter 5, we present *Metha*, a system that thoroughly tests network analysis and verification tools to find subtle bugs in their network models. Finally, in Chapter 6, we conclude, identify remaining open problems and suggest future research directions.

2 BACKGROUND

In this chapter, we provide the necessary background on how the Internet and the networks it is made up of work. In the first part (Section 2.1), we describe the most important aspects of Internet routing and forwarding. In the second part (Section 2.2), we introduce the process of network configuration and validation.

2.1 INTERNET ROUTING

For the end-users the Internet often seems like one comprehensive unit they connect to through their Internet Service Provider (ISP). In reality, however, the Internet is a complex, heterogeneous structure. It is made up of over 70 000 independently managed networks [8], so-called autonomous systems (ASes) that interconnect and exchange traffic to enable full connectivity. Each AS, in turn, consists of many devices, namely routers and switches that interconnect the end-hosts and provide them with access to the rest of the Internet.

In the following, we describe the fundamentals of how these networks and, ultimately, the Internet work. To this end, we take a bottom-up approach and start with the basic building block of the Internet and any network: IP routers (Section 2.1.1). Next, we continue explaining how routers compute the paths inside a network and then, across the networks to forward traffic from source to destination (Section 2.1.2).

2.1.1 *IP Router*

At a high level, an IP router is made up of two parts: *(i)* the control plane, which computes the network paths, and *(ii)* the data plane, which forwards the traffic along those paths. Figure 2.1 shows a simplified schematic of the inner life of a router.

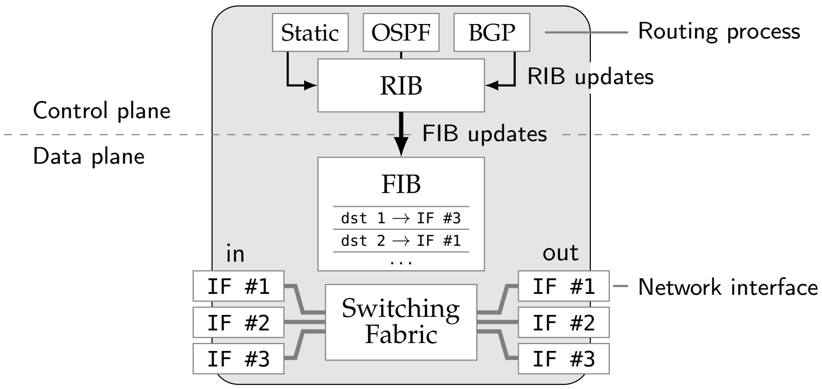


FIGURE 2.1: The high-level architecture of an IP Router, which consists of a control plane and a data plane that computes the routes and forwards the traffic, respectively.

The control plane is in charge of the routing processes, maintains the respective routing tables called routing information bases (RIBs), and selects the best routes among all routing processes. The routes are then passed to the data plane in the form of a forwarding information base (FIB) update. In addition, the control plane maintains the router's configuration, allows the network operators to modify the configuration and inspect the router's state. Typically, network operators monitor the routing tables and traffic statistics to troubleshoot the network.

The data plane is in charge of handling the network traffic. It installs the routes selected by the control plane in its forwarding information base. When it receives IP traffic on one of its interfaces, it consults the forwarding table and then sends the traffic out on the correct interface according to the best matching forwarding entry.

2.1.2 *IP Routing*

In order to compute the paths in the network and populate the forwarding tables, IP routers run one or more distributed routing protocols. The routers, which belong to the same routing domain, run in their control plane an instance of the protocol, exchange information about the network with each other and then separately compute their local state.

The computation of the paths through the entire Internet can broadly be divided into two parts: first, *intradomain routing*, which is the computation of the paths *within* a network; and second, *interdomain routing*, which is the computation of the paths across the different networks of the Internet.

Intradomain Routing

Intradomain routing is used to compute the best paths within a network, within one routing domain. The best path, which is also called the shortest path, aims to minimize the sum of the link costs from one endpoint to another. These costs are often assigned by the operator such that the computed paths satisfy management objectives: for example, operators might prefer to take the paths with the lowest delay or the highest bandwidth to optimally forward the traffic. Many intradomain routing protocols also allow to use multiple, equal-cost paths at the same time to share the load across several paths and better utilize the available resources.

Many different intradomain routing protocols exist with their respective advantages and disadvantages. These protocols can mainly be divided into two groups based on how they disseminate the routing information and how they compute the route. The two groups are *link-state* protocols and *distance-vector* protocols.

LINK-STATE PROTOCOLS In link-state protocols, all routers first build their own global view of the network by exchanging their local views and then they all independently compute the shortest paths based on this global view. To build the global view, each router floods the network with special messages, so-called Link-State Advertisements (LSAs), containing their adjacencies (to which other routers they are directly connected) and reachability information (to which IP prefixes they are attached). Based on this information, each router builds its own global view of the network locally. Then, the routers use Dijkstra's shortest path algorithm [9] to compute the shortest paths to all destinations in the network. This process continues such that in case of changes in the network due to, for example, failures or configuration updates, the paths are always updated. Typical examples of link-state protocols include Open Shortest Path First (OSPF) [10] and Intermediate System to Intermediate System (IS-IS) [11].

DISTANCE-VECTOR PROTOCOLS In distance-vector protocols, information dissemination and route computation is combined. The routers continuously exchange distance vectors, which are lists of destinations and their corresponding distance, the costs to reach them. A vector contains all destinations the sending router can reach. Whenever a router receives such a distance vector from one of its neighbors, it consults its own routing table and checks if there is any destination, which it could reach with a lower cost through that neighbor. If so, it will update its routing table entry. Then, that router sends an updated distance vector to all of its neighbors. In that way, routes propagate router-by-router through the network until all routers converge to a fixpoint. Similar to link-state protocols, distance-vector protocols will recompute the routes upon changes in the network. This process usually starts with the router closest to the change and then propagates. A typical distance vector protocol is the Routing Information Protocol (RIP) [12].

Interdomain Routing

Interdomain routing is used to compute the best paths for destinations that lie outside of one's network. Here the best path is not anymore the shortest path but depends on the different business policies and agreements across the different networks. It can very well happen that a longer path is preferred over a shorter one just because of the monetary cost.

Today, there is only a single interdomain routing protocol: the Border Gateway Protocol (BGP) [13]. BGP is a path vector protocol, meaning that each router, similar to the distance vector protocols, locally computes the best routes and passes them on to its neighbors. The difference to distance vector protocols is that the routers not only exchange destinations and the distances to reach them but also the path that the traffic will take following this route. The path represents the sequence of ASes which the route passes through. It is included to avoid routing loops as every router can check if the identifier of its own routing domain (AS number) is already on the path and reject it in that case.

Additionally, BGP differs from distance-vector protocols as it allows each operator to adapt the route selection process by specifying custom criteria. For example, by setting different local preferences for incoming routes, an operator can instruct the router to pick one or the other route.

2.2 NETWORK CONFIGURATION

In the previous section, we described the internals of the Internet and its many networks. In the following, we explain how one can configure a network such that it behaves as desired. First, we introduce common network policies, which allow operators to express the intended network behavior (Section 2.2.1). Then, we describe how one translates these intents into network configurations (Section 2.2.2). Finally, we highlight tools that help operators correctly configure and debug their network (Section 2.2.3).

2.2.1 *Network Policies*

When running a network, the operators have a clear set of requirements in mind which their network has to fulfill. These requirements are usually derived from business, reliability, and security objectives. A typical business objective, for example, is to require that the traffic is always forwarded along the cheapest path to maximize profits. Reliability objectives are often based on Service-Level Agreements (SLAs) with the network's customers, which guarantee a certain availability. Finally, security objectives might require that critical data and services are completely isolated from parts of the network and cannot be reached from the Internet.

We refer to these requirements as network policies and the set of all policies as the network's specification. The policies making up a network's specification can be coarsely divided into two main categories of policies: *data-plane* and *control-plane* policies.

The data-plane policies specify how traffic in the network has to be forwarded or when it has to be dropped. There are four typical data-plane policies in the literature [14, 15, 16, 17]: *reachability*, *isolation*, *waypointing*, and *loadbalancing*. Reachability simply requires that two endpoints can reach each other, while isolation requires the opposite as the two endpoints must not be able to reach each other. Waypointing requires traffic from one endpoint to another to pass through a given waypoint. Loadbalancing requires traffic between two endpoints to be spread across two or more paths to balance the load on the links.

The control-plane policies govern route dissemination and exchange. Examples of control-plane policies are *transit*, *route-preference*, and *feature* policies. A transit policy, for example, requires exporting routes to a neighboring network such that it can send traffic through it and reach remote destinations. Route-preference policies often implement business objectives as they allow to prefer routes with certain attributes over others. Preferences are usually used to prefer the routes advertised by a cheaper provider over those of other providers. Finally, a feature policy requires using or refraining from using certain configuration features in the network (e.g., route redistribution).

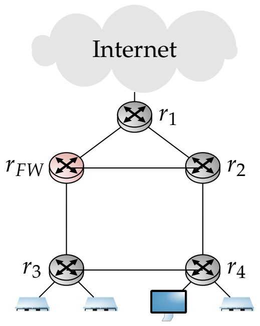
Over the years, several formal languages have been designed to capture network specifications [15, 18, 19, 20, 21, 22], which vary in their expressivity, both in the granularity at which the traffic can be controlled and in the policies supported. In theory, network operators should use these languages to unambiguously define their network's behavior both in the control plane and data plane. In practice, however, network operators often do not have a formal, well-defined network specification at hand but rather a collection of partially documented policies and objectives spread across many different design and network architecture documents.

2.2.2 *Network Configuration Process*

One of the main tasks of a network operator is to enforce the given network policies such that the network behaves accordingly. However, there is a gap between the network policies and the network's behavior. Ideally, one would simply be able to provide the network policies to the network directly. Basically, one would directly tell the network what to do. This is the vision of intent-based networking (IBN). Today, we are unfortunately still far from this vision. Instead of being able to tell the network what to do, operators have to tell the network how to do it, which is done through the network's configuration.

As an illustration, consider the network in Figure 2.2a running a link-state protocol internally. For security reasons, the operator intends to pass all incoming traffic through a firewall (r_{FW}) before delivering it in the network. To this end, the operator needs to enforce a waypointing policy. Instead of simply providing a policy, such as `policy(src: external, dst: internal; waypoint:` r_{FW}`)` to the network, the operator needs to find link weights and configure them in each router (Figure 2.2b), such that the resulting shortest paths go through the firewall (Figure 2.2c).

At a high level, a network operator takes the network policies and maps them to a corresponding network-wide configuration to achieve the de-



(a) Network topology.

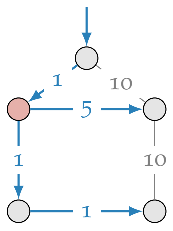
config of r_1

```
...
interface to_rFW
  ip ospf cost 1
interface to_r2
  ip ospf cost 10
...
```

config of r_{FW}

```
...
interface to_r1
  ip ospf cost 1
interface to_r2
  ip ospf cost 5
interface to_r3
  ip ospf cost 1
...
```

(b) Config excerpts.



(c) Network behavior.

FIGURE 2.2: To make traffic from the Internet pass through the firewall (r_{FW}) before reaching any internal destination, the operator has to configure the link weights such that the shortest paths include the firewall.

sired network behavior. This "indirect" process from policy to behavior is complex and makes it difficult for operators to write a correct configuration for two reasons:

1. There are many different ways to enforce a policy and achieve the intended behavior. Routers run multiple routing protocols at the same time and in many cases, one can achieve the same behavior using any of these routing protocols or combinations thereof. Additionally, the routing protocols have attributes and features that ultimately have similar effects.
2. Networks consist of tens to hundreds of routers running distributed routing protocols, which have to be individually configured such that all the routers behave correctly together. Ultimately, configuring a network is similar to programming a distributed system.

In addition, there are also operational aspects: A network is rarely run by one person. Hence, network operators have to coordinate the configuration process and the objectives of the different operators may interfere. Also, most of the time, operators have to work with existing configurations. In these cases, it is of utmost importance to ensure that existing requirements are not violated in the process of reconfiguring the network.

2.2.3 *Network Management Tools*

Due to the difficult configuration process, misconfigurations and mistakes are common. Therefore, a wide range of tools and systems has been developed to help operators safely manage their networks. These tools fall primarily into two categories: *proactive* and *reactive* tools. Proactive tools assist operators in creating correct configurations before they are deployed in the production network. Reactive tools help operators understand the behavior of their deployed configurations and debug mistakes once they have happened.

Proactive Tools

Proactive tools support network operators before applying the new or changed configuration. They allow to ensure that the configuration correctly implements a network's specification. This is done either through configuration synthesis or validation. Hence, misconfigurations and outages can be prevented before they even make their way into the production network.

CONFIGURATION SYNTHESIS Synthesis tools [14, 15, 17, 23, 24, 25] automatically generate provably correct, network-wide configurations based on the provided network specification. The existing tools vary in the supported routing protocols and the quality of the synthesized configurations. While all configurations satisfy the given specification, they differ in their size and understandability. This can make it especially difficult for operators to debug problems and apply fixes later on manually.

NETWORK ANALYSIS AND VERIFICATION Network validation tools [16, 26, 27, 28, 29, 30, 31, 32, 33, 34] allow to detect policy violations by analyzing a configuration's induced behavior in a "digital twin" of the network. These tools usually take the network-wide configuration and the policies as input and report any violations. The existing tools differ in the fidelity and type of their models. Some tools use their custom model of specific routing protocols and their features, while others rely on simulating the network with real router images [35].

HSA [36] and NetPlumber [37] rely on a geometric model to verify data plane policies. ddNF [38] proposes to compute header space equivalence classes as it is more efficient to first find the small subset of equivalence classes and then perform the analyses. Anteater [39] uses a SAT solver to verify invariants in the data plane. Batfish [26] relies on a custom Datalog model to derive the data plane given the network's configuration.

Both Bagpipe [40] and Minesweeper [16] rely on an SMT model to verify different network properties. While Bagpipe focuses only on BGP as a single protocol, Minesweeper is more general and supports a range of protocols. NV [41] is an intermediate language that allows to build network models, which can be used both to verify and simulate network control planes. ARC [27] relies on a graph representation of the control plane.

Reactive Tools

Reactive tools help operators observe and understand the current behavior of a network. There are three types of tools: first, tools that allow to observe the network's forwarding behavior; second, tools that help obtain network state (e.g., routing tables); and third, tools that allow to analyze the traffic handled by the network. The information that these tools provide is low-level. Therefore, operators themselves have to extract the important information and come up with the high-level insights.

NETWORK DIAGNOSIS UTILITIES `ping` and `traceroute` help understanding the forwarding behavior of a network. These two arcane tools allow network operators to inject traffic into the network and observe whether a particular address is reachable and the path the traffic follows. A downside of these tools is that the injected packets do not necessarily represent real traffic and can therefore only act as a proxy to understand how network traffic is handled.

ROUTING INFORMATION Looking glasses provide a snapshot of a router's routing table. Based on that, operators can understand the routes that are being advertised, chosen, and disseminated.

TRAFFIC STATISTICS Traffic statistics collected at different vantage points in the network help understanding how actual traffic is handled. Famous examples are Netflow [42] and sFlow [43]. Both rely on sampling to deal with the huge amount of traffic and report mostly aggregated statistics. The advantage of traffic statistics is that they actually represent the behavior that "real" traffic observes in the network.

3 UNDERSTANDING THE NETWORK'S FORWARDING BEHAVIOR

In this chapter, we introduce *Net2Text*, an interactive system which assists network operators in reasoning about their network's forwarding behavior.

Today, network operators spend a significant amount of their time struggling to understand how their network forwards traffic. The main difficulty lies not in the availability of the data but in the large semantic gap that separates the low-level forwarding rules from the actionable high-level insights. To understand how the network behaves, network operators need to access the network's forwarding rules, which are distributed across many different devices, and they need to correlate them with the collected traffic statistics. Bridging this gap manually (the default nowadays) is cumbersome and slow. And this is exactly where *Net2Text* helps network operators: it takes as input the operator's query expressed in natural language and the low-level network data, and automatically produces succinct natural language summaries, which efficiently capture network-wide semantics.

The main challenge behind *Net2Text* is to generate concise summaries which "explain" all the relevant behaviors, while maximizing the provided information. The key insight is to approach this as an optimization problem that aims to balance *coverage* and *explainability*. While the problem is *NP*-hard, we show that the skewness of the network forwarding state (i.e., its inherent redundancy) makes it, fortunately, well-amenable to summarization in practice. This motivates us to focus on a subspace in which every search path is of polynomial size, enabling us to design an approximation algorithm that traverses the space efficiently.

Next, we summarize our main contributions in this chapter:

- We formulate the network-wide summarization problem as an optimization problem (Section 3.3);
- we present an efficient approximation algorithm for generating high-quality summaries (Sections 3.5 and 3.6);
- and we provide an end-to-end implementation of *Net2Text*, along with a comprehensive evaluation (Section 3.9).

This chapter is organized as follows. Section 3.1 provides an overview of *Net2Text* from input to output. Section 3.2 introduces the key concepts used in this chapter. Section 3.3 formally defines the network-wide summarization problem. Section 3.4 illustrates that an exact solution to the problem does not scale. Section 3.5 presents the search space and a way to reduce it. Section 3.6 proposes ComPass, an approximation algorithm to find a good solution in the reduced search space. Section 3.7 explains the generation of the natural language summaries. Section 3.8 shows how *Net2Text* parses the operators' natural language queries. Section 3.9 evaluates our implementation of *Net2Text* and presents our findings from five interviews with network operators. Section 3.10 discusses our design choices and the extensibility of *Net2Text*. Section 3.11 reviews related work in the area. Finally, Section 3.12 concludes the chapter.

3.1 OVERVIEW

Consider a network operator wondering how the network is forwarding traffic towards Google:

> *"How is Google traffic being handled?"*

Net2Text automatically parses the question expressed in natural language and produces a concise description (also in natural language) of the current forwarding behavior observed for Google:

> *"Google traffic experiences hot-potato routing. It exits in New York (60%) and Los Angeles (40%). 66.7% of the traffic exiting in New York follows the shortest path and crosses Atlanta."*

Producing such a summary is challenging: the system has to understand what the operator is interested in, extract the relevant information, summarize it, and then translate it to natural language. Extracting this information goes beyond simply querying a database: it requires processing the data to identify common path features (e.g., the New York and Los Angeles egresses) as well as high-level features pertaining to different paths (e.g., hot-potato routing, shared waypoints). In addition, the entire process should be quick (even if the network is large) to guarantee interactivity and deal with traffic dynamics.

In the following, we give a high-level overview of how *Net2Text* manages to solve these challenges and go from the above query to the final summary using a three-stage process (see Figure 3.1).

Parsing Operator Queries in Natural Language (Section 3.8)

Net2Text starts by parsing the operator query in natural language using a context-free grammar. This grammar defines a natural language fragment consisting of multiple network features (e.g., ingress, egress, organization, load-balancing) and possible feature values (e.g., New York, Google), allowing a network operator to express a wide range of queries. Our grammar consists of ~ 150 derivation rules which are extended with semantic inference rules to infer implicit information. In the above example, our grammar infers that the operator refers to traffic destined to the *organization* Google. In addition to such summarization queries, *Net2Text* supports, at the moment, three other types of queries: *(i)* yes/no queries, "*Does* all traffic to New York go through Atlanta?"; *(ii)* counting queries, "*How many* egresses does traffic to Facebook have?"; and *(iii)* data retrieval queries, "*Where* does traffic to New York enter?". Furthermore, our grammar is extendable with new features, keywords, and names.

Net2Text maps the parsed query to an internal query language, similar to SQL. Here, the query is mapped to:

```
SELECT * FROM paths WHERE org=GOOGLE
```

This query is then run over a network database that stores the entire forwarding state of the network. Afterward, the results are passed to the core part of *Net2Text*: the summarization module.

Summarizing Forwarding States (Sections 3.3, 3.5 and 3.6)

Most queries (including the one above) can and will return a plethora of low-level forwarding entries. *Net2Text* assists the operator in reasoning about the forwarding state by automatically generating high-quality interpretable summaries out of low-level forwarding entries.

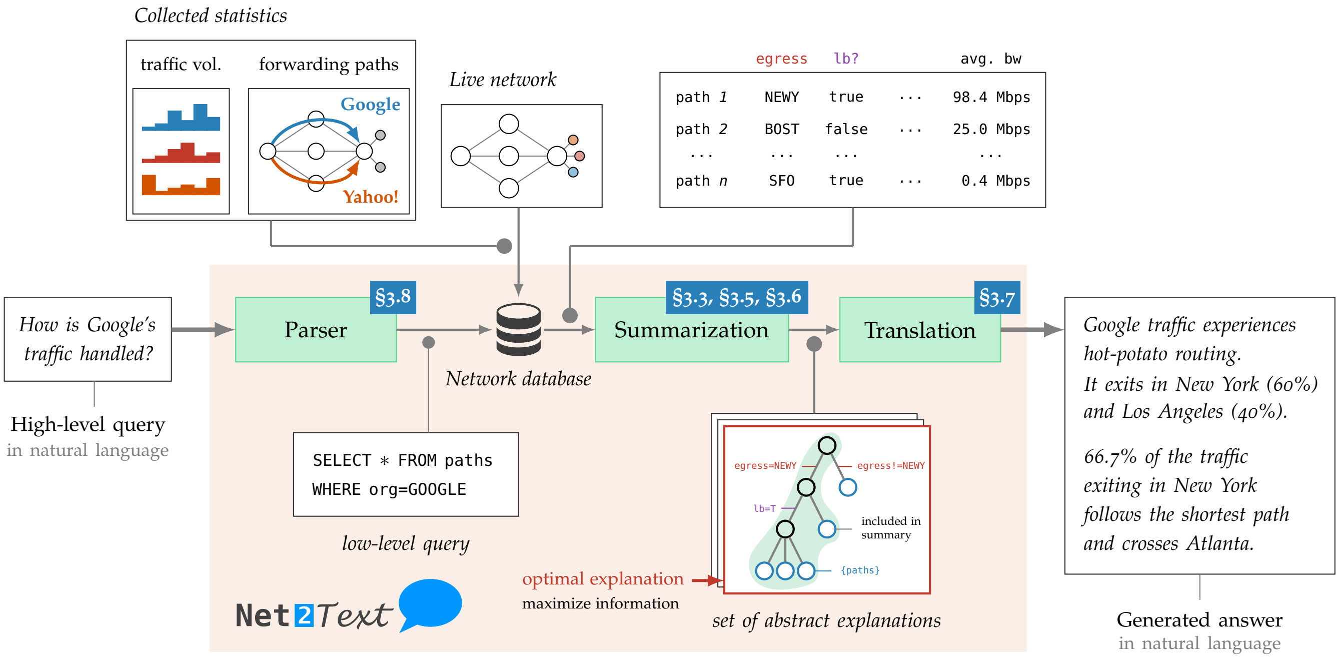
Summarizing network-wide forwarding states requires overcoming a fundamental tradeoff between *explainability* (how much detail a summary provides) and *coverage* (how many paths a summary describes). By defining a score function capturing both concepts analytically, we show that we can formally phrase this problem as an NP-hard optimization problem (Section 3.3). This renders both exhaustive techniques along with techniques based on Integer Linear Programming impractical.

To scale, we leverage the insight that traffic is skewed (heavy-tailed) across multiple levels: in the traffic distribution itself (few prefixes are typically responsible for most of the traffic [44]) or at the routing level (network topologies are usually built following guidelines, leading to repetitive forwarding patterns, e.g., edge/aggregation/core). This insight enables us to design an approximate summarization algorithm, called ComPass (Section 3.6), which explores a reduced search space that we can prove contains good summaries (Section 3.5). In addition, we show that ComPass can only summarize a sample of the forwarding entries instead of all of them with only a marginal loss in summarization quality.

Taken together, the reduced space and sampling optimization enable *Net2Text* to generate high-quality interpretable summaries for large networks (with hundreds of routers) running with full routing tables *in less than 2 seconds* (Section 3.9).

Converting Path Specifications to Text (Section 3.7)

Given a set of path specifications, *Net2Text* finally produces a summary expressed in natural language in two steps. It first extends the set with additional properties inferred by examining the specifications as a whole. For example, if the specifications imply that there are multiple paths between the egress and ingress, *Net2Text* infers that the traffic is load balanced. *Net2Text* then maps the extended specifications to sentences in natural language.



```
              egress   lb?              avg. bw
path 1        NEWY     true     ...     98.4 Mbps
path 2        BOST     false    ...     25.0 Mbps
  ...          ...      ...               ...
path n        SFO      true     ...      0.4 Mbps
```

```
SELECT * FROM paths
WHERE org=GOOGLE
```

Google traffic experiences hot-potato routing. It exits in New York (60%) and Los Angeles (40%).

66.7% of the traffic exiting in New York follows the shortest path and crosses Atlanta.

FIGURE 3.1: *Net2Text*: Workflow and key components.

3.2 PRELIMINARIES

In the following, we introduce the key terms we use in this section. We begin with the most basic building block of *Net2Text*: routing paths, which describe the traffic flows and their paths through the network. Then, we present feature functions, which simply map routing paths to a set of features. Finally, we introduce path specifications, which describe a set of routing paths and ultimately represent a summary.

ROUTING PATHS We model the network as a graph and define a network path P as a finite sequence of links. A routing path (d, P) is a pair of an IP prefix and a path, which describes that traffic to prefix d can be routed on P (a prefix can be routed on multiple paths). We denote the set of all routing paths in the network by $\mathcal{R}$.

FEATURE FUNCTIONS Routing paths have different higher-level features associated to them. For example, a routing path can follow a shortest path or can originate in a specific geographical location. Feature functions map routing paths to these higher-level features. Formally, a feature function $q\colon \mathcal{R} \to U_q$ maps routing paths $\mathcal{R}$ to *feature values* from U_q. We denote by v_q a value in U_q. We focus on the following feature functions. Organization $O\colon \mathcal{R} \to U_O$ maps every (d, P) to the organization owning d. For example, the set of organizations U_O could consist of Google, Facebook, and Swisscom. Egress $E\colon \mathcal{R} \to U_E$ maps every (d, P) to the egress of P, and ingress $I\colon \mathcal{R} \to U_I$ maps to P's ingress. Shortest path $SP\colon \mathcal{R} \to \{0, 1\}$ maps to 1 if P is a shortest path between its ingress and egress, and 0 otherwise. We use the subscripts e, i, o, and sp to denote feature values of the egress, ingress, organization, and shortest path feature functions, e.g., New York$_e \in U_E$ and $1_{sp} \in U_{SP}$.

PATH SPECIFICATIONS To explain the behavior of the network and its routing paths, we define the concept of sets of feature values called *path specifications*. Given a set of l feature functions with disjoint ranges $U_1, \ldots, U_l$[1] and a bound t (for $t \leq l$), a path specification is a (non-empty) set of feature values where the size of the set is at most t and each feature value describes a different feature function. Formally, a path specification is an element in:

$$\mathcal{S}^t_{U_1,\ldots,U_l} = \bigcup_{1 \leq m \leq t} \; \bigcup_{1 \leq j_1 < \ldots < j_m \leq l} U_{j_1} \times \ldots \times U_{j_m}$$

1 This is not a limitation, because values can be uniquely annotated.

Since the order of the feature values is not important for our needs, we treat path specifications as sets. For example, $S_{G,NY} = \{\text{Google}_o, \text{New York}_e\}$ is a path specification that contains two feature values: organization Google and egress New York.

We say a routing path (d, P) meets a path specification S, denoted as $(d, P) \models S$, if for *every* feature value in the specification $v \in S$, the corresponding feature function q maps the routing path to that value $q(d, P) = v$ if $v \in U_q$. For example, all routing paths (d, P) that are destined to a prefix d which is owned by Google and leave the network in New York, meet the specification from above.

Finally, we define a specification set $\mathcal{S}$ as a set of path specifications, i.e., $\mathcal{S} \subseteq \mathcal{S}^t_{U_1,\dots,U_l}$. A routing path (d, P) meets a specification set $\mathcal{S}$, if there exists a path specification $S \in \mathcal{S}$ such that the routing path meets that path specification, $(d, P) \models S$. For example, the following specification set $\mathcal{S} = \{\{\text{Google}_o, \text{New York}_e\}, \{\text{Seattle}_i\}\}$ captures all the routing paths that are either destined to Google and exit the network through New York or that enter in Seattle. Note that the path specifications within a specification set can overlap, which means that routing paths can meet multiple path specifications. Routing paths that enter in Seattle, are destined to Google and exit in New York, for example, meet both path specifications in the specification set above.

3.3 PROBLEM DEFINITION

In this section, we formally phrase the problem of explaining a network's forwarding behavior as an optimization problem. To explain the forwarding behavior, we aim to find a specification set that summarizes the network's routing paths. The main challenge then is to find a specification set that describes as many routing paths as possible while providing a maximal amount of information about them. To this end, we start by defining score functions that allow us to assess the quality of different specification sets. Intuitively speaking, these score functions represent the "amount of information" provided by a specification set. Given the score functions, we then formulate the problem of network summarization as a constraint optimization problem.

We phrase our optimization problem as a MAP inference task [45], in which the goal is to find an assignment that maximizes a score while satisfying a set of constraints. In our context, an assignment consists of (up to)

k path specifications, each with at most t feature values and over the feature functions $q_1, ..., q_l$. The score of an assignment is the weighted sum of the routing paths in $\mathcal{R}$ and their features described by the specification set. We define the score in two steps: first, we introduce the score of a single feature function $q \in \{q_1, ..., q_l\}$, and then, we explain how we obtain the score of all feature functions in combination.

FEATURE SCORE A score function of a feature function q maps sets of up to k specifications to a real number score:

$$\Phi_q \colon \left(\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\}\right)^k \to \mathbb{R}$$

The domain consists of specification sets, which are k-ary tuples, whose elements consist of path specifications and the empty set. The empty set $\emptyset$ denotes "no specification", and it enables us to cleanly capture specification sets with less than k specifications. To simplify definitions, we assume: $(d, P) \not\models \emptyset$ for all (d, P). For a set $\mathcal{S}$, the score $\Phi_q(\mathcal{S})$ is the weighted sum of routing paths in $\mathcal{R}$ for which q is described by a specification in $\mathcal{S}$. A path (d, P) is part of the sum if there is a specification $S \in \mathcal{S}$ containing a feature value of q that (d, P) satisfies. The weight of a path $w_{d,P}$ is a positive number (e.g., the traffic size). Formally:

$$\Phi_q(\mathcal{S}) = \sum_{(d,P)\in\mathcal{R}} w_{d,P} \cdot [\textstyle\bigvee_{S\in\mathcal{S}\colon q(d,P)\in S} (d, P) \models S] \tag{3.1}$$

In this definition, $[\cdot]$ denotes the Iverson bracket that returns 1 if the formula is satisfied or 0 otherwise.

Specification set	Φ_E	Φ_{SP}	$\Phi_{E,SP}$
$\{\{\text{NY}_e\}\}$	1	0	1
$\{\{\text{LA}_e\}\}$	2	0	2
$\{\{1_{sp}\}\}$	0	3	3
$\{\{\text{NY}_e\}, \{\text{LA}_e, 1_{sp}\}\}$	3	2	5

TABLE 3.1: Score functions for $\mathcal{R} = \{(d_1, P_1), (d_2, P_2)\}$, where $w_{d_1,P_1} = 1$ and $w_{d_2,P_2} = 2$, $E(d_1, P_1) = \text{NY}_e$ and $E(d_2, P_2) = \text{LA}_e$, and $SP(d_1, P_1) = SP(d_2, P_2) = 1_{sp}$.

Example 3.1. *Consider two routing paths $\mathcal{R} = \{(d_1, P_1), (d_2, P_2)\}$ with a traffic size of $w_{d_1,P_1} = 1$ and $w_{d_2,P_2} = 2$, respectively. The first routing path exits the* *network in New York ($E(d_1, P_1) = NY_e$), while the second one exits it in Los Angeles ($E(d_2, P_2) = LA_e$). Both routing paths follow the shortest path from ingress to egress ($SP(d_1, P_1) = SP(d_2, P_2) = 1_{sp}$). To compute, for example, the egress feature score of the specification set $\{\{NY_e\}\}$, we sum up the traffic size of all routing paths that meet this specification set when only considering the egress feature values: $\Phi_E(\{\{NY_e\}\}) = 1 \cdot 1 + 2 \cdot 0 = 1$. Similarly, we can compute the shortest path feature score of that specification set: $\Phi_{SP}(\{\{NY_e\}\}) = 1 \cdot 0 + 2 \cdot 0 = 0$. On the other hand, the shortest path feature score for the specification set $\{\{1_{sp}\}\}$ is simply the sum of the two traffic size as both routing paths follow the shortest path: $\Phi_{SP}(\{\{1_{sp}\}\}) = 1 \cdot 1 + 2 \cdot 2 = 3$. Table 3.1 summarizes the results of this example and provides scores for two additional specification sets: $\{\{LA_e\}\}$ and $\{\{NY_e\}, \{LA_e, 1_{sp}\}\}$.*

FEATURE SET SCORE A score function of a set of feature functions $q_1, ..., q_l$ maps k specifications of size at most t to a score:

$$\Phi_{q_1,...,q_l} \colon \left(\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\}\right)^k \to \mathbb{R}$$

The score is the sum of all the features' scores:

$$\Phi_{q_1,...,q_l}(\mathcal{S}) = \sum_{j \colon [1,l]} \Phi_{q_j}(\mathcal{S})$$

The last column of Table 3.1 shows the feature set score of the previous example. We can now define the optimization problem.

Definition 3.1 (Optimization Problem). *Given a set of routing paths $\mathcal{R}$, weights $w_{d,P}$ for each $(d, P) \in \mathcal{R}$, a set of feature functions $q_1, ..., q_l$, a constant k limiting the number of path specifications, and a constant t limiting the size of path specifications, we formulate the network summarization problem as:*

$$\underset{\mathcal{S} \in (\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\})^k}{\arg\max} \quad \Phi(\mathcal{S})$$

Intuitively speaking, the problem is to find a specification set $\mathcal{S}$ given a set of routing paths $\mathcal{R}$ such that the feature set score $\Phi(\mathcal{S})$ is maximized. The specification set can consist of at most k path specifications, which are of size t or smaller.

Example 3.2. *Consider 100 routing paths that are all destined to Google $\mathcal{R} = \{(Google, P_i)\}_{i=1}^{100}$ and each routing path has a weight of 1. The first 60 routing* *paths all have New York as their egress ($i \leq 60$, $E(Google, P_i) = NY_e$), whereas the remaining routing paths leave the network in Los Angeles ($E(Google, P_i) = LA_e$). In addition, the first 40 routing paths follow the shortest path ($i \leq 40$, $SP(Google, P_i) = 1_{sp}$), while the others do not. Finally, all other feature values are unique for every single routing path.*

For $k = t = 3$, an optimal solution is $\{\{NY_e\}, \{0_{sp}\}, \{NY_e, 1_{sp}\}\}$, and its score is $\Phi_E + \Phi_{SP} = 60 + 100 = 160$. The specification set $\{\{NY_e\}, \{0_{sp}\}, \{1_{sp}\}\}$ is another optimal solution. Even though, the scores are identical, the operator is likely to prefer the former specification set as it provides additional information (e.g., all traffic following the shortest path exits in New York). We leverage this insight in Section 3.5.

While this problem can be considered as a general summarization problem suitable for other contexts, the skewed nature of network traffic makes our context a better instantiation to this problem: the heavy traffic is likely to share many feature values, which can lead to solutions that are clearly better than others. At the same time, these properties are precisely the kind of information that an operator needs in order to understand the behavior of the main part of the network traffic.

3.4 EXACT SOLUTION

In this section, we show that an exact solution to the NP-hard inference problem (Definition 3.1) is (expectedly) too expensive for practical use when summarizing a large number of paths. To this end, we first show in Section 3.4.1 how to formulate the problem as an integer linear program (ILP) where the objective encodes the score function Φ and the constraints encode the path specification search space $\mathcal{S}^t_{U_1,\dots,U_l}$. Then in Section 3.4.2, we evaluate the scalability of solving this ILP using an off-the-shelf solver and show the need for an efficient, approximate algorithm, which we describe in the upcoming sections.

3.4.1 *ILP Formulation*

In the following, we explain in detail how to formulate the summarization problem as an ILP. We start by introducing all variables which encode the specification set and the features of the routing paths. Then, we explain how we encode the feature set function from before as objective

$$
\begin{aligned}
\max \sum_{(d,P)\in\mathcal{R}} \sum_{1\le i\le k} \sum_{v\in U_1\cup\ldots\cup U_l} & w_{d,P}\cdot y_{d,P,i,v} \\
\sum_{v\in U_j} x_{i,v} &\le 1 & (1)\\
\sum_{v\in U_1\cup\ldots\cup U_l} x_{i,v} &\le t & (2)\\
y_{d,P,i} - y_{d,P,v} + x_{i,v} &\le 1 & (3)\\
y_{d,P,i} + x_{i,v} - y_{d,P,i,v} &\le 1 & (4.1)\\
y_{d,P,i,v} - y_{d,P,i} &\le 0 & (4.2)\\
y_{d,P,i,v} - x_{i,v} &\le 0 & (4.3)\\
\sum_{1\le i\le k} y_{d,P,i} &\le 1 & (5)\\
x_{i+1,v} - x_{i,v} &\ge 0 & (6)\\
y_{d,P,i}, x_{i,v}, y_{d,P,i,v} &\in \{0,1\}
\end{aligned}
$$

Figure 3.2: An integer linear program for computing a specification set to explain the routing paths, where $i \in \{1,...,k\}$, $j \in \{1,...,l\}$, $(d,P) \in \mathcal{R}$, and $v \in \{U_1 \cup ... \cup U_l\}$.

function. Finally, we describe the different constraints among the variables. Figure 3.2 shows the full formulation of the ILP.

VARIABLES In our ILP formulation of the summarization problem, we have two types of variables: first, the x-variables, which encode the specification sets, and second, the y-variables, which encode the features and specifications that the routing paths meet.

For each path specification, we introduce a set of variables, one for every feature value that may be part of any path specification. Formally, we have a variable $x_{i,v}$ for every $1 \le i \le k$ and $v \in U_1 \cup ... \cup U_l$. These variables are indicator functions and range over $x_{i,v} \in \{0,1\}$. That is, if $x_{i,v} = 1$, it means that v is part of the i^{th} path specification ($v \in S_i$), otherwise v is excluded. Thus, an assignment to the x-variables uniquely defines a set of path specifications.

The y-variables encode whether a routing path meets the path specifications and which of the routing path's features are described by these path specifications. Concretely, for every routing path $(d,P) \in \mathcal{R}$, we maintain multiple binary variables:

$y_{d,P,i}$	encodes whether (d, P) meets the i^{th} specification.
$y_{d,P,v}$	indicates whether (d, P) contains a feature value of v. Note that the values $y_{d,P,v}$ are known a-priori and need not be computed during optimization.
$y_{d,P,i,v}$	encodes whether the feature v of P is described by the i^{th} specification.

These variables allow us to capture precisely in what detail a path is being described by a specification that it meets. Note that $y_{d,P,i,v}$ can be 1 only if (d, P) meets the i^{th} specification and the i^{th} specification has feature v (i.e., $y_{d,P,i} = x_{i,v} = 1$). This requirement is encoded as part of the general constraints.

OBJECTIVE FUNCTION We encode the objective function of Definition 3.1 as the weighted sum of $y_{d,P,i,v}$ variables as shown in Figure 3.2.

CONSTRAINTS The space of all path specifications is expressed as a set of constraints which states that each path specification can have at most one feature value for the same feature (constraint set (1)) and at most t features in total (constraint set (2)).

The next constraint sets encode the score function. Constraint set (3) encodes whether the routing path (d, P) meets the i^{th} specification. Intuitively, the constraints can be presented as $y_{d,P,i} \leq 1 + (y_{d,P,v} - x_{i,v})$, which means that $y_{d,P,i}$ can be 1 (to indicate that (d, P) meets the i^{th} specification) only if $y_{d,P,v} \geq x_{i,v}$ for all v, which indicate that the routing path meets all features in the i^{th} specification. Constraint set (4) encodes whether the feature value v of a routing path (d, P) is described, which may only be true if (d, P) meets the specification and the specification contains v. Lastly, the constraint set (5) guarantees that each feature value v met by (d, P) is counted only once. The total number of variables and constraints is $O(k \cdot |\mathcal{R}| \cdot |U_1 \cup ... \cup U_l|)$.

Constraint set (6) in Figure 3.2 encodes an optional requirement, which requires that every specification extends its former by at least one feature. As we have seen in Example 3.2, it can be desirable to impose such a relation between the path specifications in order to convey additional information in the summary.

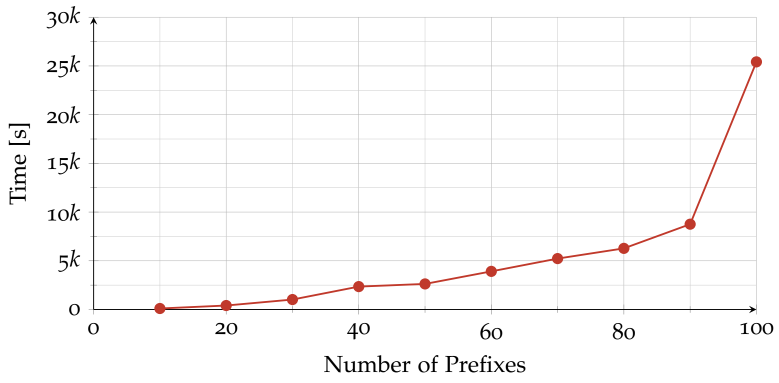


FIGURE 3.3: Running time using the ILP (optimal, but slow).

3.4.2 *ILP Scalability*

In the following, we show that an exact solution for the summarization problem is not feasible by evaluating the scalability of the ILP formulation.

We test the ILP formulation (including (6)) on the ATT NA network, which consists of 25 nodes. We generate forwarding state encompassing between 10 to 100 prefixes as described in Section 3.9.1. Figure 3.3 shows the running time for an increasing number of prefixes. The results clearly show that the running time exponentially increases. To summarize a network's forwarding state for only *100 prefixes*, the ILP already requires more than 25 000 seconds (~7h) to complete.

3.5 APPROXIMATE OPTIMIZATION

An exact solution is not feasible as we have seen in Section 3.4. Therefore, we need to come up with a scalable, approximate inference algorithm for the NP-hard summarization problem. A key challenge when designing such an algorithm is dealing with the size of the search space that is at least exponential. In our setting, we show that the search space is exponential in both t and k, making the search very challenging (Section 3.5.1). Intuitively, this stems from the fact that we need to explore two dimensions: path coverage and path explainability. To address the issue with the large search space, we leverage the fact that traffic is skewed and focus on parts of it, enabling us to trade-off expressivity of the specification set with the size of the search space. We show that the optimal solution for this part of the search space: *(i)* has at least $min\{1/k, 1/t\}$ of the score of the optimal solution for the full search space, *(ii)* the length of every search path is polynomial in t, and *(iii)* the number of children of every node is polynomial in the number of feature values (Section 3.5.2). We further identify an equivalence relation over the path specifications and leverage it to define a search space with solutions of higher quality (Section 3.5.3).

3.5.1 *An Exponential Search Space*

In this section, we analyze the size of the search space, organize the solutions in a graph, and discuss the challenges of traversing it.

SIZE OF SEARCH SPACE We begin with showing that the size of the search space is exponential in t and k. The search space is the set of all specifications, that is $(\mathcal{S}^t_{U_1,\dots,U_l} \cup \{\emptyset\})^k$. Thus, it immediately follows that its size is exponential in k. To conclude that the size is exponential in k and t, we show that the size of $\mathcal{S}^t_{U_1,\dots,U_l}$ is exponential in t. To prove this, we reduce this computation to the combinatorial problem of choosing without replacement up to t feature functions from l feature functions (we assume $l \geq t$) and then for each, picking a feature value (we assume $|U_i| \geq 2$ for all i). Then, using a combinatorial identity [46, Vol. 2, (1.37)] we get:

$$\sum_{m=0}^{t} \binom{l}{m} \cdot 2^m \geq \sum_{m=0}^{t} \binom{t}{m} \cdot \frac{2^m}{m+1} = \frac{3^{t+1}-1}{2(t+1)}$$

SEARCH SPACE AS A GRAPH We organize the solutions in a directed graph $\mathcal{G}$. The nodes of $\mathcal{G}$ are the solutions: $(\mathcal{S}^t_{U_1,\dots,U_l} \cup \{\emptyset\})^k$. There is an edge (u, v) if v extends one of u's specifications with one feature value (Figure 3.4). We distinguish between two kinds of edges: edges that extend an empty specification (colored blue) and those that extend an existing specification (colored red). Intuitively, the blue edges try to increase *coverage* by including more path specifications. This increases the number of routing paths for which the overall specification set holds. The red edges aim to increase the amount of detail captured in a path specification, resulting in better *explainability*. However, they can reduce the number of routing paths that satisfy the specification set (and thus, have the opposite effect of blue edges). We have two extreme cases in this coverage versus explainability exploration: *(i)* specification sets that maximize explainability (specifications

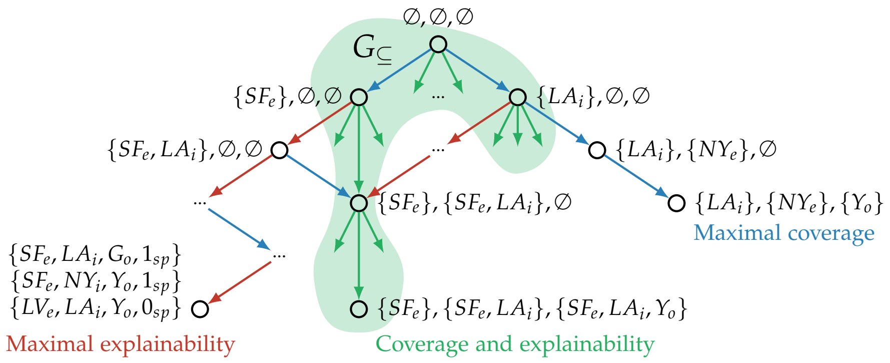

$\emptyset,\emptyset,\emptyset$
$G_{\subseteq}$
$\{SF_e\},\emptyset,\emptyset$
$\{LA_i\},\emptyset,\emptyset$
$\{SF_e, LA_i\},\emptyset,\emptyset$
$\{LA_i\},\{NY_e\},\emptyset$
$\{SF_e\},\{SF_e, LA_i\},\emptyset$
$\{LA_i\},\{NY_e\},\{Y_o\}$
Maximal coverage
$\{SF_e, LA_i, G_o, 1_{sp}\}$
$\{SF_e, NY_i, Y_o, 1_{sp}\}$
$\{LV_e, LA_i, Y_o, 0_{sp}\}$
$\{SF_e\},\{SF_e, LA_i\},\{SF_e, LA_i, Y_o\}$
Maximal explainability
Coverage and explainability

FIGURE 3.4: Part of the search space for k=3 specifications, t=4 feature values per specification using the features egress *(e)*, ingress *(i)*, organization *(o)* and shortest path *(sp)*.

are of size t) and *(ii)* specification sets that maximize coverage (all specifications are of size 1). Depending on the weights and number of routing paths, the optimal solution sits in-between these two extremes.

Example 3.3. *$\{\{New\ York_e\}\}$ maximizes coverage, as it is very general by only fixing a single feature and its value, while $\{\{New\ York_e, Dallas_i, Google_o, 1_{sp}\}\}$ maximizes explainability, as it is more specific by fixing four features and their values.*

SEARCH CHALLENGE An important ingredient in any search strategy is the solution scoring function, which guides the search towards the optimal result, while effectively pruning subspaces. In our setting, such a score function is even more critical as the size of the search space is exponential in k and t. An immediate candidate for a score function is Φ, as in Definition 3.1. However, Φ can guide us towards a good solution only if we restrict our traversal to nodes reachable through the blue edges. This is due to a monotonicity property guaranteeing that if v is reachable from u only through blue edges, then $\Phi(v) > \Phi(u)$ (since v includes all feature values described by u). However, the red edges do not have this property for Φ (as it trades off path coverage with explainability). Even if we consider a different scoring function, pruning is unlikely to be effective and the traversal may end up exploring an exponential number of nodes. Instead, we consider a reduced subspace that has shorter paths and satisfies the monotonicity property for *every type of edge*.

3.5.2 *A Reduced Search Space*

In this section, we define a reduced space $\mathcal{G}_{\subseteq}$, which is a subspace of $\mathcal{G}$. Our reduced space leverages the fact that traffic is skewed, and thus the heavy part of the traffic shares many feature values. This means that specifications consisting of these common feature values have higher scores than other specifications and that these higher-scored specifications intersect. This motivates us to focus only on solutions whose specifications are contained in one another. Such an approach guarantees that the solutions balance path coverage (provided by the shorter specifications) and explainability (provided by the larger specifications). We show that $\mathcal{G}_{\subseteq}$ contains solutions which are not significantly worse than an optimal solution in $\mathcal{G}$. Specifically, we show that $\mathcal{G}_{\subseteq}$ contains a solution whose score is at least $min\{\frac{1}{k}, \frac{1}{t}\}$ of an optimal solution in $\mathcal{G}$, in the worst case. In $\mathcal{G}_{\subseteq}$, the size of the search paths is t (instead of $t \cdot k$ as in $\mathcal{G}$), and every node has at most $\sum_{i=1}^{l} |U_i|$ children (instead of $k \cdot \sum_{i=1}^{l} |U_i|$).

The nodes of $\mathcal{G}_{\subseteq}$ are all specification sets whose path specifications are *extensions of one another*. More formally, a node has the property that its (non-empty) specifications can be ordered to $S_1, ..., S_m$ such that: *(i)* the path specifications are subsets of each other, $S_1 \subset ... \subset S_m$; and *(ii)* the size of them continuously increases, for all $1 \leq i \leq m$, $|S_i| = i$. For example, $\{\{\text{New York}_e\}, \{\text{New York}_e, 1_{sp}\}\}$ is a node in $\mathcal{G}_{\subseteq}$, as the second path specification extends the first one with 1_{sp}, while the specification set $\{\{\text{New York}_e\}, \{\text{Los Angeles}_e\}\}$ is not part of $\mathcal{G}_{\subseteq}$.

The edges of $\mathcal{G}_{\subseteq}$ combine both kinds of edges of $\mathcal{G}$. Concretely, there is an edge (u, v) if v contains *all* specifications of u and *also* contains a specification that extends the largest specification of u with an additional feature value. More formally, if the (non-empty) specifications of u are ordered as defined before to $S_1, ..., S_m$, then v has the specifications $S_1, ..., S_m, S_{m+1}$ such that $S_m \subset S_{m+1}$ and $|S_{m+1}| = m + 1$. Figure 3.4 highlights the nodes of $\mathcal{G}_{\subseteq}$ with a green background and shows the edges of $\mathcal{G}_{\subseteq}$ (which are different from the edges of $\mathcal{G}$) in green.

OPTIMALITY We now discuss how solution optimality in $\mathcal{G}_{\subseteq}$ relates to that in $\mathcal{G}$. Intuitively, there are two "worst case scenarios". First, if the optimal specification set consists only of path specifications that are of size t, a solution of $\mathcal{G}_{\subseteq}$ that contains any such specification contains subsets of this specification as well, which "take the spot" of the other specifications, *without necessarily contributing to the score*. To illustrate this, consider the scenario where $k = 3$, $t = 4$ and there are 3 paths, p_1, p_2, p_3 with weight 1 whose feature values are $\{e_1, i_1, o_1, sp_1\}$, $\{e_2, i_2, o_2, sp_2\}$,$\{e_3, i_3, o_3, sp_3\}$, respectively (where e_n is an egress, i_n is an ingress, o_n is an organization, and sp_n is an indicator for shortest path). An optimal solution is to pick exactly these three combinations of feature values as path specifications resulting in an overall score of 12. However, in $\mathcal{G}_\subseteq$, a solution that includes one of these path specifications contains also its subsets, making the score of the optimal solution only 3. The other "worst case scenario" is if all optimal solutions are of size 1. In this case, path specifications of size greater than 1 may only slightly increase the score. To illustrate this, consider the scenario where $k = 3$, $t = 4$ and there are 12 paths, $p_1, \dots, p_{12}$ with weight 1 such that $p_1, \dots, p_4$ have property e_1, $p_5, \dots, p_8$ have property e_2 and $p_9, \dots, p_{12}$ have property e_3 (besides this, there are no common feature values). An optimal solution is $\{e_1\}$, $\{e_2\}$, $\{e_3\}$ whose score is 12. However, because of the structure of our space, the optimal solution has a score of 6.

The following lemma states that the maximum gap between the scores of the optimal solution in $\mathcal{G}_\subseteq$ and $\mathcal{G}$ is at most a factor of $min\{\frac{1}{t}, \frac{1}{k}\}$.

Lemma 3.1. *Let $OPT_\mathcal{G}, OPT_{\mathcal{G}_\subseteq}$ be the optimal solutions in $\mathcal{G}$ and $\mathcal{G}_\subseteq$. Then,* $\min\{\frac{1}{t}, \frac{1}{k}\} \cdot \Phi(OPT_\mathcal{G}) \leq \Phi(OPT_{\mathcal{G}_\subseteq})$.

PROOF SKETCH Denote the optimal solution as the following specification set: $\{x^1_1,, ..., x^1_{t_1}\}$, $\dots$,$\{x^k_1, ..., x^k_{t_k}\}$. By the score definition and since $t_i \leq t$, $\forall i, j. \frac{j}{t}\Phi(\{x^i_1,, ..., x^i_{t_i}\}) \leq \Phi(\{x^i_1, ..., x^i_j\})$. Without loss of generality, assume that $\{x^1_1,, ..., x^1_{t_1}\}$ has the highest score. Then, $\Phi(\{x^1_1,, ..., x^1_{t_1}\}) > OPT/k$. We distinguish two cases.

1. If $k \leq t_1$, then the score of $\{x^1_1\}, \{x^1_1, x^1_2\}, ..., \{x^1_1, ..., x^1_k\}$ is at least $k/t \cdot (OPT/k)$. Since $\{\{x^1_1\}, ..., \{x^1_1, ..., x^k_1\}\}$ is a node in $\mathcal{G}_\subseteq$, the claim follows.
2. Otherwise, if $t_1 < k$, then $\{\{x^1_1\}, ..., \{x^1_1,, ..., x^1_{t_1}\}\}$ is a node in $\mathcal{G}_\subseteq$ and since $\Phi(\{x^1_1,, ..., x^1_{t_1}\}) > OPT/k$, the claim follows.

3.5.3 *A Path Equivalent Space*

In this section, we define a search space which is similar to $\mathcal{G}_\subseteq$ but may contain solutions with higher score. Intuitively, this is obtained by "merging" nodes in $\mathcal{G}_\subseteq$ that are equivalent with respect to the routing paths which are covered by the nodes. In other words, for every two nodes in this space, there is at least one path satisfying one but not the other. Path equivalence does not imply the same score. For example, if $\{e_1\}, \{i_1\}$ are path equivalent, then $\{e_1, i_1\}$ is also path equivalent to them, but with a score that is twice as high as them. This is because each path contributes its weight twice, once per feature. By considering only nodes that are not path equivalent, we can potentially obtain better solutions, without sacrificing the lower bound of Lemma 3.1.

We use this observation to modify $\mathcal{G}$ to a space $\mathcal{G}_=$ whose solutions consist of specifications that are *(i)* contained in one another (like $\mathcal{G}_\subseteq$) and *(ii)* maximal with respect to path equivalence. In our example, this means that $\{e_1\}, \{i_1\}$ are not part of any solution in $\mathcal{G}_=$, but $\{e_1, i_1\}$ might be if its extensions are not equivalent to it. In $\mathcal{G}_=$, there is an edge (u, v) if, for u whose specifications are $S_1 \subseteq ... \subseteq S_m$, we have *(i)* the specifications of v are $S_1, ..., S_m, S_{m+1}$, *(ii)* $S_m \subset S_{m+1}$, and *(iii)* for any subset S such that $S_m \subset S \subset S_{m+1}$, S_{m+1} and S are path equivalent. By construction, $\mathcal{G}_=$ has solutions which are at least as good as those in $\mathcal{G}_\subseteq$, which gives us:

Lemma 3.2. *Let $O_{\mathcal{G}_\subseteq}$ and $O_{\mathcal{G}_=}$ be optimal solutions in $\mathcal{G}_\subseteq$ and $\mathcal{G}_=$, respectively. Then, $\Phi(O_{\mathcal{G}_=}) \geq \Phi(O_{\mathcal{G}_\subseteq})$.*

By traversing $\mathcal{G}_=$, algorithms can return solutions with larger path specifications than the ones obtained by traversing $\mathcal{G}_\subseteq$. This follows since the maximal size of a specification in $\mathcal{G}_\subseteq$ is k, while the size of specifications in $\mathcal{G}_=$ is up to t.

3.6 THE COMPASS ALGORITHM

We now introduce ComPass, an algorithm to **c**ompute **p**ath **s**pecification**s** by traversing the search space $\mathcal{G}_=$.

ComPass (Algorithm 3.1) lazily computes nodes in $\mathcal{G}_=$ and continues to the node with the highest increase in score. It takes as input a set of routing paths $\mathcal{R}$, a set of feature functions $q_1, \ldots, q_l$, and constants k and t denoting the maximal number of specifications and the maximal size of each path specification. ComPass starts by initializing the set of solutions $\mathcal{S}$ and the current specification L to the empty set, and $\mathcal{Q}$ to the set of all candidate feature functions (Lines 1–3). In up to k iterations, the best feature value is selected to extend L–namely, the feature value that maximizes the score of

Algorithm 3.1: ComPass ($\mathcal{R}, q_1, \ldots, q_l, k, t$)

```
Input  : $\mathcal{R}$: a set of routing paths.
         $q_1, \ldots, q_l$: a set of feature functions.
         $k$: limit on the number of specifications.
         $t$: limit on the size of specifications.
Output: A set of specifications $\mathcal{S}$.
1  $\mathcal{S} = \emptyset$                                   // The specification set
2  $L = \emptyset$                                   // The last computed specification
3  $\mathcal{Q} = \{q_1, ..., q_l\}$                                   // Candidate features
4  while $|\mathcal{S}| < k$ do
5      $q, v = \arg\max_{q \in \mathcal{Q}, v_q \in \mathcal{U}_q} \sum_{(d,P) \in \mathcal{R}} w_{d,P} \cdot [q(d,P) = v_q]$
6      $L = L \cup \{v\}$
7      $\mathcal{Q} = \mathcal{Q} \setminus \{q\}$
8      $\mathcal{R} = \mathcal{R} \setminus \{(d,P) \mid q(d,P) \neq v\}$
9      if $|L| = t$ then $\mathcal{S} = \mathcal{S} \cup \{L\}$; break
10     while $\exists v \in \mathcal{U}_{\mathcal{Q}}.(L \cup \{v\} \equiv L)$ do
11         $L = L \cup \{v\}$
12         if $|L| = t$ then $\mathcal{S} = \mathcal{S} \cup \{L\}$; break
13         $\mathcal{Q} = \mathcal{Q} \setminus \{q\}$
14     $\mathcal{S} = \mathcal{S} \cup \{L\}$
15 return $\mathcal{S}$
```

$\mathcal{S}$ as defined by the score function (Eq. (3.1)) when adding it to L. This can be formalized as maximizing the function on Line 5.

Let v be this feature value and q its feature. Then, L is extended with v and q is dropped as L cannot contain another feature value from $\mathcal{U}_q$. The paths in $\mathcal{R}$ not meeting v are dropped as well, as these will not be described by the next specifications (Lines 6–8). Then, if the size of L reaches the bound t, the loop breaks as it is impossible to extend L further (Line 9). Otherwise, ComPass computes the maximal specification that is equivalent to L by checking whether it can be extended with other feature values (Lines 10–13). Finally, L is added to $\mathcal{S}$ (Line 14), and the next iteration begins. To ensure the limit of t is not exceeded, once L has reached this bound, ComPass completes and returns the current specification sets. This means that ComPass may return fewer than k specifications. It can be shown that this solution has a higher score than a solution with k specifications that are not representative. Intuitively, this follows since the paths described by the descendants are subsumed by the paths described by their ancestors.

Example 3.4. *Consider $\mathcal{R} = \{(Google, P_i)\}_{i=1}^{100}$, each with weight 1, and $k = t = 2$. As before, we assume that* (i) *if $i \leq 60$, $E(Google, P_i) = NY_e$, and $E(Google, P_i) = LA_e$ otherwise,* (ii) *for $i \leq 40$, $SP(Google, P_i) = 1_{sp}$, and* (iii) *all other feature values are unique for every path. We now show how ComPass computes the optimal solution $\{\{NY_e\}, \{NY_e, 1_{sp}\}\}$. In its first iteration, ComPass discovers that the feature value NY_e maximizes the score. It thus extends L to $\{NY_e\}$, prunes the egress feature E from $\mathcal{Q}$, and removes from $\mathcal{R}$ all paths whose egress is not New York. Since $\{NY_e\}$ is the representative of its equivalence class, it is added to $\mathcal{S}$. In the second iteration, the feature value 1_{sp} maximizes the score. Hence, ComPass extends L with 1_{sp}. Since the limit $t = 2$ has been reached, the loop breaks (Line 7), and the specification set $\{\{NY_e\}, \{NY_e, 1_{sp}\}\}$ is returned.*

FINDING THE BEST FEATURE VALUE To avoid iterating every feature value separately in Line 5 (which can incur high overhead), we find the best feature value by iterating over the feature functions in $\mathcal{Q}$ and the routing paths in $\mathcal{R}$ and storing the score of each feature value in a hash table. Then, with a single pass over the hash table, we find the feature value with the highest score.

GUARANTEES Our next theorem states that ComPass computes a solution whose score is at least $\frac{1-f}{1-f^{\min\{t,k\}}}$ of the optimal solution in $\mathcal{G}_=$, where $f \in (0, 1)$ is the maximal portion of paths that a child of a node can have. Note that since ComPass explores $\mathcal{G}_=$, whose nodes are not path equivalent, f cannot be 1.

Theorem 3.1. *Given that there is $f \in (0, 1)$ such that for every pair of path specifications A, A' if $A \subset A'$, then $\Phi(\{A\}) \leq f \cdot \Phi(\{A'\})$. Then, if O is the solution returned by ComPass, we have $\frac{1-f}{1-f^{\min\{t,k\}}} \cdot OPT_{\mathcal{G}_=} \leq O$.*

PROOF SKETCH Let the optimal solution of the inference problem be $OPT = \{\{a\}, \{a, b\}, ..., \{a, b, ..., m\}\}$ and the specification set that ComPass returned be the specification set $S_{ComPass} = \{\{a'\}, \{a', b'\}, ..., \{a', b', ..., m'\}\}$. By the assumption, $\Phi(\{a, b\}) \leq f \cdot \Phi(\{a\}) \leq f \cdot \Phi(\{a'\})$. By induction, $\Phi(\{a, b, ..., j\}) \leq f^{|\{a,...,j\}|} \cdot \Phi(\{a'\})$. Since the length of the largest specification in OPT is $\min\{k, t\}$, the length of the optimal solution is at most $\Sigma_{1 \leq j \leq \min\{k,t\}} f^j \cdot \Phi(\{a\}) = \frac{1-f^{\min\{t,k\}}}{1-f} \cdot \Phi(\{a\})$. By the greedy operation, we have $\Phi(\{a\}) \leq \Phi(\{a'\})$. Since $\Phi(\{a'\}) \leq \Phi(S_{ComPass})$, we get $\Phi(\{a\}) \leq \Phi(S_{ComPass}) \leq \frac{1-f^{\min\{t,k\}}}{1-f} \cdot \Phi(\{a\})$, which means that ComPass is a $\frac{1-f}{1-f^{\min\{t,k\}}}$-approximation algorithm.

Example 3.5. *The factor f is determined by the pair of nodes $A \subset A'$ whose scores are the closest. In the previous example with $k = t = 2$, $A = \{\{NY_e\}, \emptyset\}$, $A' = \{\{NY_e\}, \{NY_e, 1_{sp}\}\}$. Since $\Phi(A) = 60$ and $\Phi(A') = 100$, we get that $f = 0.6$. By the theorem, ComPass returns a solution whose score is at least 62.5% of the optimal solution in $\mathcal{G}$.*

SPEEDING UP COMPASS BY SAMPLING To compute the best feature value, ComPass iterates in Line 5 over all routing paths. This step is very expensive, especially if the number of routing paths and feature functions is large. To mitigate this problem, we leverage two observations that allow ComPass to uniformly sample the routing paths instead of considering all routing paths. First, Internet traffic is heavily skewed, which means that most traffic is directed towards a few organizations (e.g., CDNs), and egresses see different traffic volumes depending on the peering. This means that sampling is likely to pick representative routing paths. Second, by the score function definition, optimal solutions consist of specifications describing the main part of the traffic. This means that specifications representing little traffic have little effect on the decisions ComPass makes. This implies that sampling will perform well as it is more likely to ignore the specifications with few routing paths than the ones with many.

3.7 FROM SPECIFICATIONS TO SUMMARIES

In this section, we describe how *Net2Text* produces a natural language summary given a specification set $\mathcal{S}$ (generated by ComPass). It begins by augmenting $\mathcal{S}$ with additional information in three steps. It then transforms the path specifications in $\mathcal{S}$ to natural language sentences using templates. In the first two steps, *Net2Text* augments $\mathcal{S}$ with information computed as a byproduct by ComPass (i.e., additional specification sets and the amount of traffic). In the third step, *Net2Text* extends $\mathcal{S}$ with high-level features, which cannot be directly computed by ComPass. We next describe these steps and illustrate them on our example, $\mathcal{S} = \{\{\mathrm{NY}_e\}, \{\mathrm{NY}_e, 1_{sp}\}\}$.

STEP 1: ADDING PATH SPECIFICATIONS *Net2Text* extends every $S \in \mathcal{S}$ with the next m (a parameter) best path specifications that have the same parent in $G_=$ and are values of the same feature function. These path specifications can be extracted from the computation of ComPass (in Line 5). In our example, for $m = 1$, this step results in adding $\{\mathrm{LA}_e\}$ to $\mathcal{S}$ as NY_e and LA_e have the same parent and same feature function (egress).

This will eventually be translated to a single sentence: *Google traffic exits in New York and Los Angeles.*

STEP 2: ADDING TRAFFIC SIZE Then, *Net2Text* extends every $S \in \mathcal{S}$ with the total weight of the paths it describes to let the operator understand how much traffic the summary covers. In our example, this gives $\{(60\%, \{\text{NY}_e\}), (40\%, \{\text{LA}_e\}), (40\%, \{\text{NY}_e, 1_{sp}\})\}$.

STEP 3: COMPUTING HIGH-LEVEL FEATURES Next, we extend $\mathcal{S}$ with high-level features (e.g., load-balancing, waypointing, or hot-potato routing) that are not properties of single paths but rather of *sets of paths*, i.e., entire specifications. Thus, these features can only be identified after ComPass computed the best specification set. Each of these high-level feature comes with a set of criteria that the routing paths in a specification have to meet for the high-level feature to hold. For example, for load-balancing, the ingress and egress of a specification have to be fixed and the paths described by it need to be disjoint. In our example, *Net2Text* inferred that a common waypoint for $(40\%, \{\text{New York}_e, 1_{sp}\})$ is Atlanta as all the paths in this specification go through Atlanta, and thus this specification is extended to $(40\%, \{\text{New York}_e, 1_{sp}, \text{Atlanta}_w\})$. In addition, *Net2Text* inferred that the traffic to Google experiences hot-potato routing as it has multiple egresses and all the traffic is forwarded to the closest one.

STEP 4: TRANSLATION TO NATURAL LANGUAGE Finally, $\mathcal{S}$ is translated to natural language sentences. The sentences are a composition of multiple basic templates. To create fluency in the summary, *Net2Text* connects related sentences by building upon the previous one. In addition, it does not repeat information. For example, the second sentence in our example summary in Section 3.1 does not repeat that it refers to Google traffic, and the percentage shown is relative (i.e., $40\%/60\% = 66.7\%$). Namely, $\{(40\%, \text{NY}_e, 1_{sp}, \text{Atlanta}_w)\}$ is mapped to: *66.7% of the traffic exiting in New York follows the shortest path and crosses Atlanta.*

3.8 PARSING QUERIES

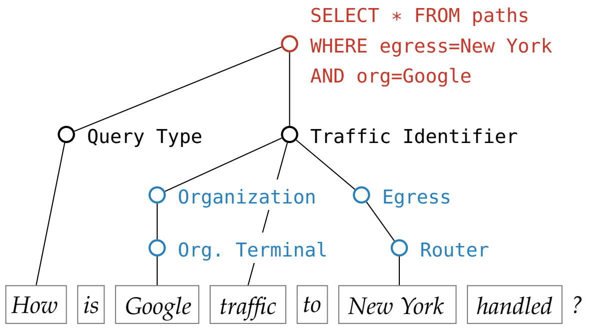
To leverage *Net2Text*'s summarization capabilities, the operator needs to provide the feature functions Q, t, k, and the routing paths $\mathcal{R}$. Typically, once Q, t and k are specified, the operator queries the network database to obtain $\mathcal{R}$. To simplify this, *Net2Text* allows the operator to submit queries in natural language which it then translates to SQL-like queries for the network database. In the following, we describe how *Net2Text* parses these queries expressed in natural language.

NETWORK GRAMMAR The grammar consists of rules specifying how constituents of the queries (e.g., clauses, words) can be composed. The rules also specify the semantics of the constituents (e.g., the network terms such as "ingress"), which enables the parser to construct the SQL-like query. The grammar consists of two parts: *(i)* a structural part ($\sim$ 70 rules), which defines the allowed constituent compositions; and *(ii)* a domain-specific part consisting of mapping rules ($\sim$ 80 rules), which capture the network specific features (e.g., egress, organization) as well as keywords (e.g., router and location names). This split enables the operator to easily extend the grammar with new features and keywords without having to deal with the structure of the queries.

STRUCTURAL GRAMMAR This grammar defines the query structure and its building blocks. We identify two main building blocks: query type and traffic identifier. Depending on the query type, there may be additional building blocks. There are four query types: yes/no ("Is/Does..."), counting ("How many..."), data retrieval ("What is/are..."), and explanation ("How is/does traffic..."). The query type determines whether the answer is yes/no, a count, a list, or a summary (obtained using ComPass). The traffic identifier defines the WHERE clause of the SQL-like query. The attributes selected by the query are either determined by the query (in data retrieval queries) or are simply a wildcard (i.e., $\star$).

Figure 3.5 illustrates the structural parsing. *"How"* defines the desired behavior of *Net2Text* (summarize the data with ComPass), while "Google traffic to New York" is the traffic identifier. The black rules (1-3) are part of the structural grammar, while the blue rules (4 & 5) are part of the domain-specific grammar, which we discuss next.

DOMAIN-SPECIFIC GRAMMAR This grammar defines a mapping between keywords and names to features and their values. For example, the grammar defines the rules *(i)* "to $N \rightarrow$ egress=N" indicating that the natural language phrase "to N" means that N is a name of an egress, where N is a non-terminal and *(ii)* $N \rightarrow$ NY, LA, ..., lists the possible egress names. Using these rules, "to NY" is parsed to `egress=NY` in the SQL-like query.



```
             SELECT ∗ FROM
Q[SEM=(                    )] -> How TI[SEM=?ti]              (1)
            paths WHERE ?ti
                TI[SEM=(?pc)] -> PC[SEM=?pc]                  (2)
      PC[SEM=(?pc1 and ?pc2)] -> PC[SEM=?pc1] PC[SEM=?pc2]    (3)
          PC[SEM=(egress=?n)] -> To N[SEM=?n]                 (4)
                 N[SEM=(NY)] -> New York                      (5)
```

Figure 3.5: A parse tree and rules in the network grammar.

3.9 EVALUATION

In this section, we evaluate *Net2Text*'s scalability and usability by addressing the following research questions:

RQ1 How does *Net2Text* scale to different networks when summarizing all routing paths? We show that *Net2Text* can summarize large forwarding state in few seconds only (Section 3.9.2).

RQ2 How does sampling affect the quality of *Net2Text*'s summaries? We show that *Net2Text* generates high-quality summaries whose scores are within 5% of the unsampled summary (Section 3.9.3).

RQ3 How useful is *Net2Text* for network operators? Several network operators confirmed in interviews that virtual assistants like *Net2Text* are indeed useful (Section 3.9.4).

RQ4 Is *Net2Text* practical? We show *Net2Text*'s end-to-end implementation in a case study using a live, virtual network (Section 3.9.5).

3.9.1 *Methodology*

We run our Python-based prototype ($\sim 3k$ lines of code) on a machine with 24 cores at 2.3 GHz and 256 GB of RAM. For the experiments, we implemented an ISP-like forwarding state generator, which we use to produce realistic forwarding state for various Topology Zoo [47] topologies ranging from 25 to 197 nodes (Table 3.2). The generator enables us to control how "summarizable" a state is by varying how skewed it is.

FORWARDING STATE GENERATION Our generator synthesizes network-wide forwarding states (i.e., the set of routing paths $\mathcal{R}$) for a given number of IP prefixes and a given network topology in five consecutive steps. *First*, it randomly chooses a set of egress nodes (see Table 3.2). *Second*, it creates a prefix-to-organization mapping by relying on the CAIDA AS-to-organization dataset [48]) and a full IPv4 RIB [49]. *Third*, for each organization, it chooses the number of egresses using an exponential distribution fitted according to real measurements [50], after which the actual egresses are uniformly chosen from the set of egress nodes. *Fourth*, for each node, it computes its forwarding state by picking for each prefix the closest egress. *Fifth*, each routing path (d, P) is finally associated with an amount of traffic sampled from an exponential distribution. This leads to few organizations owning many prefixes, carrying relatively more traffic than others (as shown in [44]). The generator can also generate extra features whose values are arbitrarily picked.

GENERALITY While we generate the input forwarding state, we stress that our results are representative because: *(i)* the scalability of ComPass does *not* depend on the actual feature values but only on the number of features (see Section 3.6); and *(ii)* the quality analysis does not depend on the actual score but rather on the ratio compared to other scores under the same setting.

3.9.2 *Scalability Analysis*

We evaluate *Net2Text*'s scalability by measuring the time it takes to summarize all routing paths (worst-case) while varying the number of key dimensions: prefixes, nodes, and feature functions. To evaluate the sampling optimization of ComPass, we run ComPass four times: without path sampling and with a sampling rate of $1/10$, $1/100$, $1/1\,000$. We repeat each experiment 10 times and report median results.

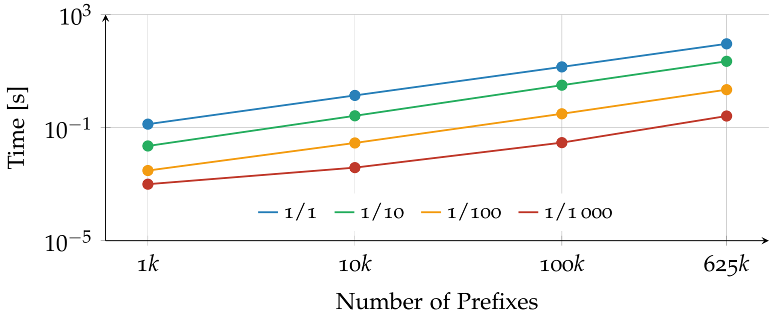


FIGURE 3.6: Time as a function of the number of prefixes and sampling rate.

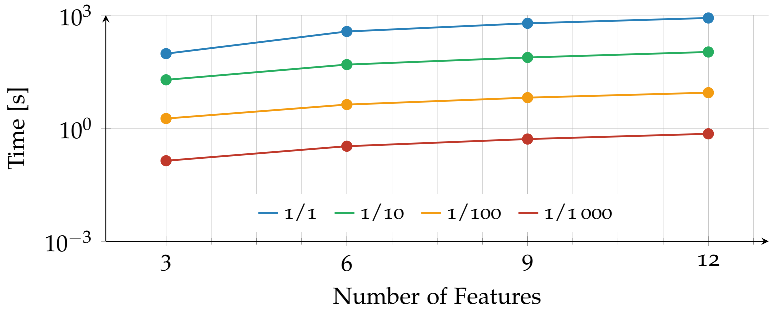


FIGURE 3.7: Time as a function of the number of features and sampling rate.

Figure 3.6 shows the results when varying the number of prefixes from 10^3 to 10^5 and the full RIB for the ATT NA topology using 3 feature functions. The results indicate that *Net2Text* scales linearly in the number of prefixes. The running time decreases proportionally to the sampling rate. Without sampling, summarizing forwarding states with 625k prefixes takes about 100 seconds and *less than one second* with a sampling rate of 1/1 000. Figure 3.7 shows a similar trend when varying the number of features from 3 to 12 and using a full RIB.

Table 3.2 shows the runtime of *Net2Text* when considering different topology sizes, with full routing tables (625k prefixes) and 3 feature functions. The table reports the runtime both with (rate of 1/1 000) and without sampling. We see that the runtime is roughly linear in the number of nodes in the network. More importantly, our results indicate that *Net2Text* scales to large networks with hundreds of nodes thanks to sampling: it takes less

Topology	Nodes	Egresses	No sampling	1/1 000
ATT NA	25	10	94.07 s	**0.26 s**
Switch	42	15	128.12 s	**0.24 s**
Sinet	74	30	223.91 s	**0.50 s**
GTS CE	149	40	611.81 s	**1.18 s**
Cogent	197	50	766.61 s	**1.84 s**

TABLE 3.2: With sampling (Section 3.6), *Net2Text* summarizes large network forwarding states (> 600k prefixes), *within 2 seconds*, for networks with close to 200 nodes.

than 2 seconds for *Net2Text* to summarize the entire forwarding state of the Cogent topology.

3.9.3 *Quality Analysis*

We now evaluate the effect of sampling the input data (i.e., the forwarding paths) and show that doing so only marginally impacts the quality of the summary. In addition, we show that ComPass compares well against two baselines both in terms of quality and running time.

We measure the quality of a summary using the score function presented in Section 3.3. Intuitively, the score represents the traffic volume of the paths covered by the resulting summary, rewarding more detailed summaries by multiplying the volume of each path by the level of details (i.e., number of feature values) present in the summary. When computing the score, we always account for all entries that match the resulting summary and not just for the sampled entries. As in Section 3.9.2, we consider the problem of summarizing every single entry in the network database.

For the experiment, we generate forwarding state for the ATT NA topology with a full routing table (> 600k prefixes) and vary the sampling rate from 1 to 1/5 000 000. Note that we have more entries in the network database than the total number of prefixes in the routing table as there is at least one path from every node to every prefix. Hence, even with sampling rates higher than the number of prefixes, we still have paths to summarize. For this setup, we have more than 15 million entries in the network database.

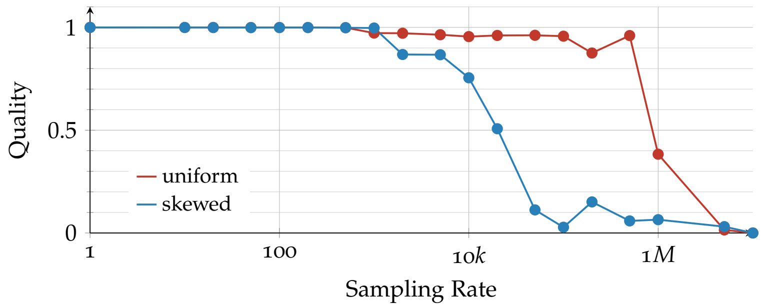


FIGURE 3.8: Summary quality as a function of sampling rate.

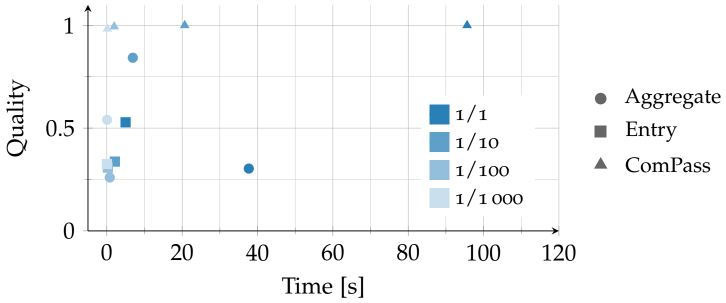


FIGURE 3.9: *Net2Text* produces summaries of higher quality than two simple baselines.

Figure 3.8 shows the score of the summary for different sampling rates normalized to the score without sampling. We ran the experiment for two different scenarios: *(i)* highly skewed traffic distributions among the feature values, where the size difference between the feature values is high; and *(ii)* uniform distributions, where the difference between them is low. Our results show that the sampling rate at which the score of the summary drops significantly is very high. Even with sampling rates of 1/1 000, ComPass still creates summaries whose qualities are within 5% of the unsampled summary.

To further illustrate the quality of the summaries obtained using ComPass, we compare it against two baselines. Both baselines iterate once over the relevant routing paths and pick the most detailed specification (e.g., $\{\text{New York}_e, \text{Philadelphia}_i, \text{Google}_o\}$). From this specification, we build the full specification set by randomly removing one feature value after the other to obtain, for example, the following specification set: $\{\{\text{New York}_e\}, \{\text{New York}_e, \text{Google}_o\}, \{\text{New York}_e, \text{Philadelphia}_i, \text{Google}_o\}\}$. The baselines differ in how they choose the most detailed specification: *Entry*, takes the routing path with the highest weight and uses it as the most detailed specification; and *Aggregate*, aggregates all routing paths with the same feature values and uses the largest aggregate. When computing the quality of the summaries, we consider all routing paths matching the resulting summary. Thanks to sampling, ComPass produces *higher* quality summaries in the same amount of time as the two baselines (see Figure 3.9). If we also consider the information added to the summary by extending it as described in Section 3.7, we see that ComPass outperforms the baselines by almost 5 times.

3.9.4 *Usefulness*

To better assess the usefulness of virtual assistants in the network setting in general and *Net2Text* in particular, we conducted five interviews with network operators of different ISP networks (research, Tier 1 and Tier 2) and one enterprise network. We discussed with them four aspects: *(i)* the need for virtual assistants, *(ii)* the usefulness of an natural language interface, *(iii)* the value of natural language output, and *(iv)* the usefulness of *Net2Text* and the queries it supports.

ASPECT 1: NEED FOR VIRTUAL ASSISTANTS All operators see opportunities for virtual assistants in all tasks that require to process a lot of data. They can imagine to offload and automate the identification and extraction of the relevant information, which they can then use to draw their conclusions for further action. A virtual assistant allows them to focus on remedying, rather than identifying and analyzing the event.

ASPECT 2: RELEVANCE OF THE NATURAL LANGUAGE INPUT The possibility to write queries in natural language was well-perceived. One operator mentioned that a natural language interface could simplify the onboarding of new employees. Some operators, however, do not mind a fixed query language or writing their own scripts.

ASPECT 3: RELEVANCE OF THE NATURAL LANGUAGE OUTPUT Most operators see value in natural language summaries as they are concise and simple to understand, especially for less technical persons. Depending on the query, some operators mentioned that they would like to see visualizations of the summary (e.g., a graph) in addition to text.

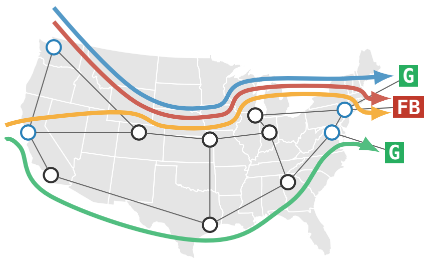
ASPECT 4: USEFULNESS OF NET2TEXT QUERIES All of the operators confirmed that the queries currently supported by *Net2Text* are relevant. In particular, they appreciated the ability to query about incoming traffic. In addition, most operators testified interest in service-oriented queries, instead of purely destination-oriented ones (e.g., traffic to the Gmail-service instead of Google traffic in general).

In the discussions, we saw a clear difference between the queries of ISP and enterprise network operators. While the ISP operators were mostly concerned about where traffic was entering and leaving the network, the enterprise operator was more interested in the status of the different applications running in the network and their policies (e.g., is there always a firewall on the path).

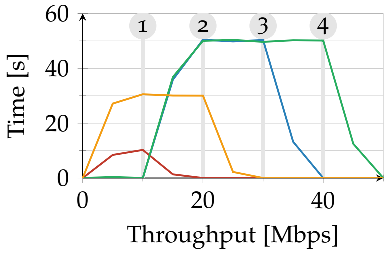
3.9.5 *Case Study*

We showcase our end-to-end implementation of *Net2Text* by running it in a virtual network using the Internet2 topology (Figure 3.10a). For the routers, we use the popular Quagga software routing suite [51]. Routers in Seattle, Sunnyvale, New York and Washington are connected to external peers. The router in New York receives routes to both Google and Facebook, while the router in Washington only receives routes to Google. All external routes have the same local-preference. We generate transit traffic entering via Seattle and Sunnyvale towards both destinations. The flows are highlighted in Figure 3.10a and the measured throughput is depicted in Figure 3.10b. Every ten seconds, *Net2Text* collects the entire forwarding state and the traffic statistics to summarize the current network-wide forwarding behavior as indicated by the four labels.

Figure 3.10c shows the 4 summaries produced by *Net2Text*. We see that *Net2Text* is able to explain the current forwarding behavior at different levels of detail and automatically zooms in on the largest part of the traffic. At the time of the second summary, for example, traffic for Google has spiked (green and blue) and is now three times larger than Facebook. We see that *Net2Text* automatically focuses on the traffic to Google and provides more details about it, yet it still mentions traffic to Facebook. In the third summary, we see how *Net2Text* captures higher-level constructs that are not directly present in the database such as "hot-potato routing" (Section 3.7).



(a) Internet 2 Topology.



(b) Network Throughput.

1. *"Traffic has a single egress (New York), and goes to a single destination (Facebook). It enters at the following ingresses: Sunnyvale (76%) and Seattle (24%)."*
2. *"Traffic goes to the following destinations: Google and Facebook. Traffic for Google exits through Washington (50%) and New York (50%)."*
3. *"Traffic is destined to Google. It experiences hot-potato routing. It exits through the following egresses: New York (50%) and Washington (50%)."*
4. *"Traffic leaves through Washington, has a single ingress (Sunnyvale), and goes to Google."*

(c) Resulting summaries.

FIGURE 3.10: We ran our *Net2Text* implementation in a live network emulating Internet 2 (a) and vary the network throughput according to (b). The resulting summaries at the different points in time accurately capture the network's forwarding behavior.

3.10 DISCUSSION

In this section, we discuss our decision to use a natural language interface, possibilities to extend *Net2Text* and ways to obtain and build the underlying network database.

WHY NATURAL LANGUAGE? We believe that a chat-like interface provides a familiar and intuitive way for operators to interact with their network. That said, our summarization contribution is useful in its own right, independently of the NLP interface. For example, as an illustration, we could easily translate *Net2Text* summaries to a graph-based representation (e.g., using PGA [21]) rather than natural language.

WHAT ABOUT NEW FEATURE FUNCTIONS? While we only deal with a limited set of features in this chapter, we stress that ComPass is flexible and can deal with *any* features defined over routing paths. Additional features (e.g., the TCP port number) can easily be added by adding a new field to the database. For the translation, the singular and plural of the feature name also have to be added to the rules. The operator can also add a mapping of feature values to some string, e.g., TCP port 80 to HTTP.

WHAT ABOUT THE NETWORK DATABASE? The problem of building the network database is orthogonal to the problem of explaining the network-wide forwarding behavior. Therefore, we assume that the network database is fed with high-quality *and* consistent data and focus on the problem of summarizing it. This is a strong assumption. However, gathering high-quality state consistently is challenging and the quality of our summaries will inevitably suffer should the data be incomplete, outdated, or inconsistent. Fortunately, multiple works have looked at the problem of extracting network data in a fast and consistent manner, which *Net2Text* can directly leverage. In particular, Libra [29] tackles the problem of capturing consistent snapshots of the network forwarding state. Similarly, FlowRadar [52] and Stroboscope [53, 54] tackle the problem of quickly gathering fine-grained traffic statistics.

3.11 RELATED WORK

In this section, we describe related work in the area of network provenance, which aims to explain why a network state occurs, and in the area of network management using natural language, which allows network operators to manage their network through a natural language interface.

NETWORK PROVENANCE *Net2Text*'s high-level objectives of *explaining how networks behave* bear similarities with many works on Network Provenance (e.g., [55, 56, 57, 58, 59, 60, 61]). The main difference between these works and *Net2Text* is that *Net2Text* does not aim at explaining *why* a particular state is observed (by following the derivation history), but rather summarizing *what* is the current state being observed to make it understandable to human operators. *Net2Text* can therefore be seen as complementary to these frameworks. Once network operators understand what the network behavior is, they can then ask questions about why. We also believe that *Net2Text*'s summarizing capabilities can be applied to summarize provenance explanations which often tend to be large.

CONNECTING NATURAL LANGUAGES AND NETWORKS The idea of using natural language to manage networks has been around for quite some time. Most work in this area provides a natural language interface in order to control the network and its behavior, while *Net2Text* focuses on monitoring the network and explaining its behavior. A general problem of using natural language is intent disambiguation. In contrast to query languages, network operators can specify the same intent in many different ways or specify different intents using very similar natural language utterances. Therefore, it is important to correctly infer the intents, especially when changing a network's state based on that intent.

Alsudais et al. [62] introduce NLP techniques to control and query the state of SDN networks. However, unlike *Net2Text*, their work does not provide any abstraction capability and is limited to simple yes/no questions and answers along with simple control tasks such as rate limiting a flow. Similarly, Esposito et al. [63] design a natural-language interface to manage Software-Defined Infrastructures (SDIs). The intents are specified in the Gherkin language, a restricted language that is easy for operators to understand but structured enough such that the commands can unambiguously be parsed.

Lumi [64] is a system that allows network operators to configure their network using natural language. From the operators' natural language instructions, Lumi extracts the important information and assembles an intent in their own specification language, which is called Nile (Network Intent LanguagE) [65]. Finally, the intent is compiled into network configuration commands expressed in Merlin [20]. To successfully disambiguate the operator's intents, Lumi always confirms with the operators that it understood their intent correctly before proceeding.

INFERRING NETWORK INTENTS FROM THE FORWARDING STATE The Anime framework [66] also addresses the problem of extracting high-level insights from a network's forwarding state. It differs from *Net2Text* in several ways. First, Anime does not aim to be a virtual assistant for the network operator and does not provide a natural language interface like *Net2Text*. Second, Anime introduces two objective measures to assess the quality of an inferred intent: *(i)* precision, which measures whether the intent covers any unobserved paths; and *(ii)* recall, which measures whether the intent covers all observed paths. While Anime aims to find intents with a recall of 100% that maximize the precision, *Net2Text* does the opposite by finding intents with a precision of 100% that maximize the recall. Third, Anime's specification language is more expressive as it allows for multiple feature types, such as hierarchical features. And fourth, due to the richer specifications, Anime is significantly slower than *Net2Text*.

3.12 CONCLUSION

In this chapter, we introduced *Net2Text*, a system to assist network operators in understanding their network's forwarding behavior. *Net2Text* is based on efficient summarization techniques which generate interpretable summaries (in natural language) out of the low-level forwarding state and traffic statistics. *Net2Text* makes several key contributions: *(i)* a precise formulation of the network-wide summarization problem as an optimization problem; *(ii)* ComPass, an approximate algorithm for generating high-quality summaries, which scales to large data sets; *(iii)* a thorough experimental evaluation illustrating that *Net2Text* can summarize the network-wide forwarding behavior of hundreds of routers carrying full routing tables within 2 seconds.

4 UNDERSTANDING THE NETWORK'S CONFIGURATION

In this chapter, we introduce *Config2Spec*, a system which helps network operators understand their network's configuration by automatically mining all the policies the configuration enforces.

Today, network operators have a range of network validation and configuration synthesis tools at their disposal. These tools help them prevent human-induced downtimes caused by misconfigurations. While useful in theory, these tools have, unfortunately, not yet been widely deployed in practice [7]. A reason for that is the requirement to provide a full specification of the correct intended network behavior. Writing down the precise specification is a daunting task. Surely, network operators can try to manually work their way through the network's configuration. However, this is challenging and error-prone: the configurations have been written over years and years by a team of network engineers (some of which do not even work there anymore), and they have been "polluted" by short-term fixes to problems that needed immediate attention (e.g., congestion and hardware problems) and have not been removed ever since. *Config2Spec* supports network operators in this task by automatically mining the network's specification. It takes the network-wide configuration and a failure model (e.g., up to k failures) as input and returns the precise specification: *all* policies that hold under the failure model and *only* those.

The main challenge behind *Config2Spec* lies in exploring two exponential search spaces: *(i)* the space of all possible policies, and *(ii)* the space of all possible network-wide forwarding states. The key insight is to address specification mining with a combination of data-plane analysis and control-plane verification. *Config2Spec*, first, prunes the large space of policies by analyzing a few forwarding states using data-plane analysis and then validates the remaining policies with control-plane verification.

Next, we summarize our main contributions in this chapter:

- We present a novel approach to automatically mine a network's specification using a combination of data-plane analysis and control-plane verification (Section 4.3);

- we present three domain-specific techniques to improve *Config2Spec*'s effectiveness: policy-aware sampling, policy grouping and topology-based trimming (Sections 4.4–4.6);
- and we provide an end-to-end implementation of *Config2Spec*, along with an extensive evaluation across different topologies and baselines (Section 4.7).

The rest of this chapter is structured as follows. Section 4.1 defines the specification mining problem and presents two baseline approaches. Section 4.2 provides an overview of the entire system. Section 4.3 presents the predictor which dynamically switches between data-plane analysis and control-plane verification. Section 4.4 introduces data-plane analysis and policy-aware sampling. Section 4.5 introduces control-plane verification and policy grouping. Section 4.6 presents topology-based trimming. Section 4.7 evaluates our implementation of *Config2Spec*. Section 4.8 reviews related work in the area. Finally, Section 4.9 concludes the chapter.

4.1 PROBLEM DEFINITION

In this section, we define the problem of mining a network specification. To this end, we first introduce the two key concepts used throughout this chapter: *network specifications*, which are composed of a set of policies, and *failure models*, which specify under which failures the network specification should hold. Then, we introduce the network specification mining problem and discuss two baseline approaches together with their shortcomings, thus motivating our solution.

RUNNING EXAMPLE Throughout this chapter, we refer to the example shown in Figure 4.1. Here, we have a network that consists of five routers and seven links. There are two host networks, `p1` and `p2`, attached to routers `1` and `2`. All routers are in the same OSPF area, and the OSPF weights are depicted on the links. An IP access control list (ACL) on the interface from router `5` to `2` drops all packets destined to prefix `p1`.

FAILURE MODELS A failure model consists of a *symbolic environment* and a number k. The symbolic environment defines which links are up or down, and which links may fail. Technically, a symbolic environment is a partition of the network links L into three subsets L_{up}, L_{down}, and $L_{symbolic}$ (i.e., given L_{up} and L_{down}, we can derive $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$). The number k is a bound on the total number of links which can be simultaneously

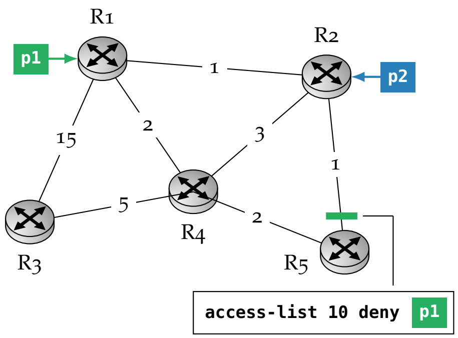


FIGURE 4.1: An OSPF network with five routers and two destinations. An ACL at router 5 blocks traffic destined to prefix p1, attached to router 1.

down. A *concrete environment* is a partition of the network links L into two subsets L_{up} and L_{down}. Namely, all links are fixed to a concrete state: up or down. We say that a failure model with a symbolic environment L^{SE}_{up}, L^{SE}_{down}, $L^{SE}_{symbolic}$ and a bound k, captures a concrete environment with L^{CE}_{up} and L^{CE}_{down} if $L^{SE}_{up} \subseteq L^{CE}_{up}$, $L^{SE}_{down} \subseteq L^{CE}_{down}$, and $|L^{CE}_{down}| \leq k$. Intuitively, a failure model captures all concrete environments for which the links in L^{SE}_{up} are up, the links in L^{SE}_{down} are down, and there are at most k links which are down.

Example 4.1. *Consider the following failure model for our running example: $L_{symbolic} = L$ (i.e., L_{up} and L_{down} are the empty sets) and $k = 1$. This model describes any concrete environment with at most one link failure. There are eight concrete environments which meet this failure model: one where no link is down, and seven in which each of the links fails once. Another failure model is $L_{up} = \{$2-4$\}$, $L_{down} = \{$2-5$\}$, $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$, and $k = 2$. This model describes any concrete environment whose link between routers 2 and 4 is up, the link between 2 and 5 is down, and the rest may be up or down. Since $k = 2$, another failed link is allowed in addition to 2-5. There are six concrete environments that meet this failure model.*

NETWORK SPECIFICATION AND POLICIES A *network specification* consists of a set of policies. A *policy* captures a specific behavior in the network (e.g., reachability of two routers). It is modeled with a predicate (a constraint) which, given a concrete environment, evaluates to true if the policy holds for that concrete environment, and false otherwise. For our running example, the `reachability(5, p2)` policy evaluates to true for the

Policy	Meaning
`reachability(r, p)`	Traffic from `r` can reach `p`.
`isolation(r, p)`	Traffic from `r` is isolated from `p`.
`waypoint(r, w, p)`	Traffic from `r` to `p` passes through `w`.
`loadbalancing(r, p)`	Traffic from `r` to `p` is load balanced on at least two paths.

TABLE 4.1: Network policies (`r` and `w` are routers, `p` is a prefix).

concrete environment in which all links are up, and to false for the concrete environment where all links are down. We say a policy holds for a failure model if it holds for all concrete environments captured by the failure model. For example, the policy `reachability(5, p2)` holds for the failure model $L_{symbolic} = L$ and $k = 1$, but not for $k = 3$ as `p2` is not reachable for `5` anymore when both the links between `2` and `5`, and between `4` and `5` fail.

In our work, we focus on reachability, isolation, waypointing, and load balancing policies (summarized in Table 4.1). The reachability, isolation, and load balancing policies are defined as predicates over a router `r` and a subnet in the network `p`. These evaluate to true if, for the given concrete environment, traffic from router `r` can reach the prefix `p`, is isolated from `p`, or load balanced on at least two paths to `p`, respectively. The waypoint policy is defined over two routers `r` and `w`, and evaluates to true if, for the given concrete environment, traffic from `r` destined to prefix `p` passes through `w`. We note that our approach is extensible to any policy that is defined over the forwarding state (e.g., equal length paths).

PROBLEM DEFINITION We now define the problem of mining a network specification:

Given a network configuration and a failure model, mine the network specification, i.e., the set of all policies which hold under the failure model.

For our running example and the failure model $L_{symbolic} = L$ and $k = 1$ (modeling up to one link failure), the network specification consists of the following ten policies:

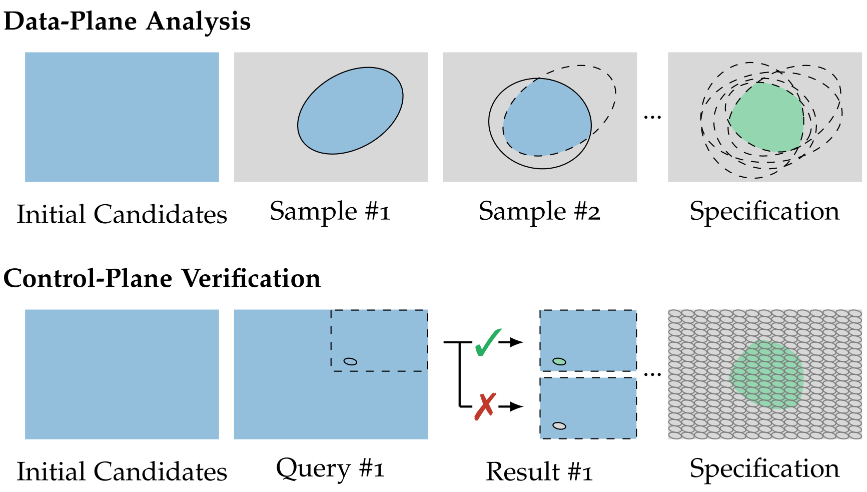


FIGURE 4.2: Illustration of the baseline approaches.

```
reachability(1, p1),   reachability(1, p2),
reachability(2, p1),   reachability(2, p2),
reachability(3, p1),   reachability(3, p2),
reachability(4, p1),   reachability(4, p2),
reachability(5, p2),   loadbalancing(4, p2).
```

BASELINE SOLUTIONS To address the above problem, one may consider two baseline approaches: *(i)* data-plane analysis and *(ii)* control-plane verification.

DATA-PLANE ANALYSIS Data-plane analysis tools (e.g., Batfish [26]) enable reasoning of policies that hold for a specific concrete environment. Today, such tools are scalable enough to reason about all of our considered policies within seconds or minutes (mostly depending on the size of the network). Thus, one could use such tools to mine a specification by iterating over all concrete environments captured by the failure model, computing a data plane for each (from the configuration), and analyzing them to infer the set of policies that hold for each concrete environment. The solution is then the intersection of all obtained policy sets. Figure 4.2 (top) visualizes this approach. Initially, every policy is a candidate which can be part of the network specification (blue area). Then, with every sampled data plane, the set of policies that hold for it are computed (shown in the circle). These are then intersected with the policies of the previous samples (dashed circles). In the end, the remaining candidate policies are those that hold for all samples and thus form the network specification (green area). Unfortunately, for large topologies or failure models with many concrete environments, this approach does not scale (see Section 4.7.2).

CONTROL-PLANE VERIFICATION Control-plane verification tools (e.g., Minesweeper [16]) enable checking individual policies for a given failure model. Technically, this can be accomplished by symbolically encoding the network, its configuration, the failure model and a policy into a formula, and then checking the satisfiability of this formula. Figure 4.2 (bottom) visualizes this approach. Initially, all policies are part of the set of candidates of the specification. At every step, one policy (circle) is picked and posed as a query to the verifier. The verifier either returns that the policy holds (green) or shows a counterexample to disprove it (gray). In the end, every policy has either been verified or disproved. As in data-plane analysis, while control-plane verification tools scale to the policies that we consider, enumerating all possible policies and checking them one by one in the above manner is prohibitive (see Section 4.7.2).

4.2 OVERVIEW

In this section, we first present our key insight of combining the two baseline approaches from Section 4.1 and explain the reasoning behind it. Then, we provide an overview of the entire system.

4.2.1 *Key Insight*

We address the problem of mining a network specification by combining the baseline approaches and leveraging their respective strengths: data-plane analysis is efficient at pruning policies, while control-plane verification is efficient at validating policies. The key idea of our combination is to reduce the space of policies by sampling forwarding states and pruning policies using data-plane analysis, and then running control-plane verification to verify a small set of remaining policies.

This combination works well because many policies which do not hold are *dense violations*. That is, they are violated for many of the concrete environments captured by the failure model. For example, in our running example and the failure model $L_{symbolic} = L$ with $k = 1$ (up to one failure), the policy `waypoint(3, 1, p2)` only holds for the concrete environment in which all links are up, but the one from router `3` to `4`. Thus, by sampling any other concrete environment (e.g., $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$), and computing all policies that hold for it, we can prune `waypoint(3, 1, p2)`.

On the other hand, there are *sparse violations*, which are policies that do not hold for the failure model, but are violated only by very few concrete environments. For example, in our running example and the same failure model, the policy `isolation(5, p1)` is violated only by two concrete environments: *(i)* $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$ and *(ii)* $L_{down} = \{$`1-2`$\}, L_{up} = L \setminus L_{down}$. Unless we check these particular environments, this policy cannot be pruned by data-plane analysis. Thus, we prune sparse violations during the step of control-plane verification. Since the overall number of true policies and sparse violations is often significantly smaller than the number of concrete environments, control-plane verification is an efficient solution for this.

4.2.2 *The* Config2Spec *System*

We build on this insight to design *Config2Spec* (Figure 4.3), which takes as input the network configuration (of all devices) and a failure model and outputs the network specification.

Config2Spec runs in a loop which dynamically switches between the two approaches until the specification is mined. To achieve this, *Config2Spec* relies on three main components: *(i)* predictors, *(ii)* data-plane analysis, and *(iii)* control-plane verification. In addition, *Config2Spec* maintains two sets of policies, `cands` which overapproximates the specification, and `verified` which underapproximates it. We next explain these sets, the algorithm flow and the three components. We show the full algorithm of *Config2Spec* in Algorithm 4.1.

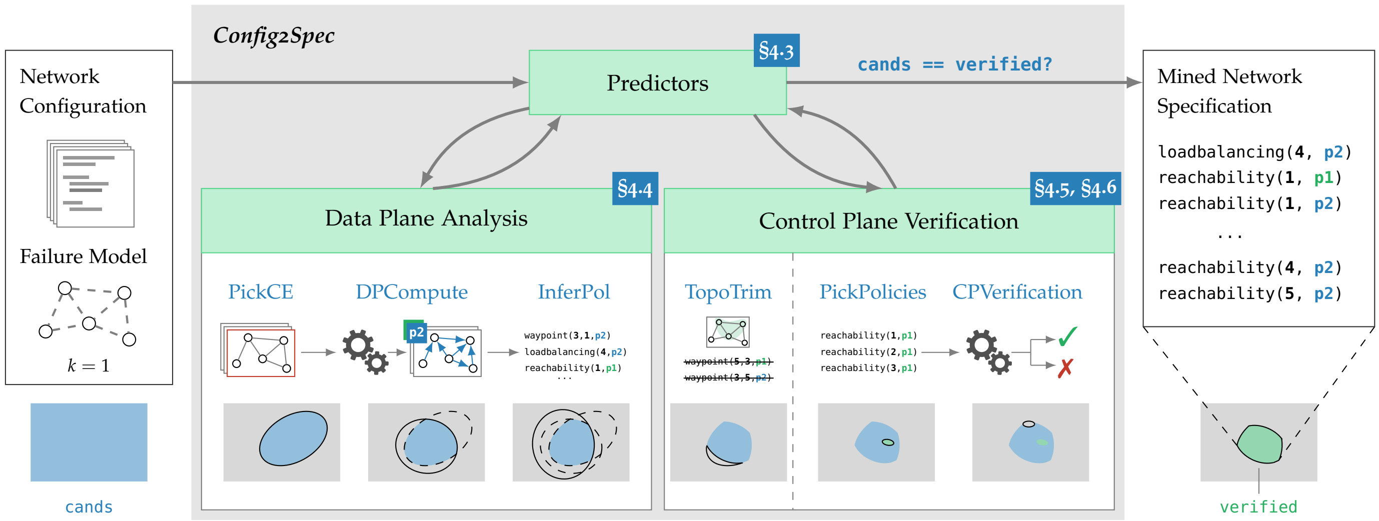


FIGURE 4.3: *Config2Spec* mines the specification from the network configuration and the failure model. It relies on three components: predictors, data-plane analysis, and control-plane verification. It maintains two sets: `cands`, consisting of the current candidate policies, and `verified`, consisting of the verified policies. During the execution, policies are removed from `cands` or added to `verified`. When `cands` equals `verified`, both equal the network specification, and then `verified` is returned.

Algorithm 4.1: *Config2Spec*(conf, $\mathcal{F}$)

Input : conf: The network configuration.
$\mathcal{F}$: the failure model (i.e., L_{up}, L_{down}, $L_{symbolic}$, k).

Output: verified: the set of all policies that hold for the given configuration and failure model.

```
cands ← $allPolicies()$
verified, prevEnvs, lastFwds ← $\emptyset, \emptyset, \emptyset$
$T^{TP}_{verify}, T^{RT}_{verify}$ ← initVerificationTimes()
$T^{TP}_{analysis}, T^{RT}_{analysis} \leftarrow 0, 0$
totalEnvs ← $\sum_{j=0}^{k} \binom{|L_{symbolic}|}{j}$
while $cands \neq verified$ do
    DP-RT ← $T^{RT}_{analysis} \cdot$ (totalEnvs − |prevEnvs|)
    CP-RT ← $T^{RT}_{verify} \cdot$ |cands \ verified|
    if $T^{TP}_{analysis} < T^{TP}_{verify}$ or $DP\text{-}RT < CP\text{-}RT$ then
        env ← PickCE($\mathcal{F}$, cands\verified, prevEnvs, lastFwds)
        lastFwds, $T^{RT}_{analysis}$ ← DPCompute(env, conf)
        pols = InferPol(lastFwds)
        $T^{TP}_{analysis} \leftarrow (\text{cands} \setminus \text{pols} = \emptyset)\ ?\ \infty : \frac{T^{RT}_{analysis}}{|\text{cands} \setminus \text{pols}|}$
        cands ← cands ∩ pols
        prevEnvs ← prevEnvs ∪ {env}
        if $|prevEnvs|$ = $totalEnvs$ then verified ← cands
    else
        pols ← PickPolicies(cands, verified)
        cex, $T^{RT}_{verify}$ ← CPVerification(pols, conf,$\mathcal{F}$)
        if $cex = \perp$ then
            verified ← verified ∪ pols
            $T^{TP}_{verify} \leftarrow \frac{T^{RT}_{verify}}{|\text{pols}|}$
        else
            cands ← cands \ {$p$ ∈ pols| cex violating $p$}
            $T^{TP}_{verify} \leftarrow \frac{T^{RT}_{verify}}{\{p \in \text{pols} |\ \text{cex violating}\ p\}}$
return verified
```

CANDS AND VERIFIED *Config2Spec* keeps two sets: *(i)* `cands`, containing the current candidate policies, i.e., the policies that are known to hold or have not been pruned yet, and *(ii)* `verified`, containing the policies that are known to hold. `cands` initially contains all possible policies (blue area in Figure 4.3), while `verified` is initially empty (green area in Figure 4.3). We note that in practice, to avoid storing all policies in `cands`, only to prune many of them upon the first iteration of data-plane analysis, *Config2Spec* directly initializes `cands` to the set of policies that holds for some concrete environment.

An invariant of the execution is that `cands` is a superset of the network specification, i.e., it contains at least all the policies that hold, while `verified` is a subset of it, i.e., it contains only policies that hold. *Config2Spec* terminates when these sets are equal – implying both equal the network specification – and then returns `verified`. Precision is ensured as *Config2Spec* does not miss any policy thanks to the invariant that `verified` contains only true policies (no false positives), while `cands` cannot miss a true policy (no false negatives).

FLOW At each iteration, *Config2Spec* checks if `cands` equals `verified`. If so, it terminates. Otherwise, it checks two predictors to decide which approach is the more promising one to pursue: data-plane analysis or control-plane verification.

PREDICTORS (SECTION 4.3) We design two predictors to heuristically estimate which approach is likely to be more effective and dynamically transition between them. The predictors consider the execution times and the number of pruned and verified policies. The first predictor checks the effectiveness of each approach in classifying policies by measuring the time it needs to classify a single policy. The second predictor estimates the remaining time to mine the full specification.

DATA-PLANE ANALYSIS (SECTION 4.4) In every iteration of data-plane analysis, *Config2Spec* samples a concrete environment, computes the policies that hold for it, and removes from `cands` any other policy. To sample a concrete environment, it executes `PickCE`, which employs a novel policy-aware sampler to find a concrete environment likely to prune more policies. Then, *Config2Spec* computes the data plane of that sample via `DPCompute`, which relies on existing analysis tools (e.g., Batfish [26]). Next, it executes `InferPol` to compute all policies which hold for this data plane, and updates `cands` accordingly. Finally, *Config2Spec* checks whether all data planes have been analyzed. If so, it sets `verified` to `cands`, as the entire failure model has been covered and the full specification has been mined.

control-plane verification (Section 4.5) In each iteration of control-plane verification, *Config2Spec* verifies a set of policies. For this, *Config2Spec* first executes `PickPolicies` to pick the next set of policies to verify. It then calls `CPVerification`, which relies on existing verification tools (e.g., Minesweeper [16]). The verifier either determines that all policies hold or returns a counterexample. In the former case, *Config2Spec* adds all the policies to `verified`, while in the latter case *Config2Spec* removes the ones violated by the counterexample from `cands`. Before the first iteration of control-plane verification, *Config2Spec* invokes `TopoTrim` to reduce the verification overhead.

topology-based trimming (Section 4.6) `TopoTrim` analyzes the topology and the failure model to trim (i.e., prune) policies which cannot hold regardless of the configuration (e.g., due to a lack of connectivity). It relies on graph algorithms to prune `reachability`, `waypoint`, and `loadbalancing` policies.

4.3 config2spec's predictors

In this section, we describe how *Config2Spec* dynamically decides whether to run the data-plane analyzer or the control-plane verifier. This decision relies on two predictors that capture the effectiveness of the approaches and the expected time remaining. Accordingly, *Config2Spec* infers which approach is more likely to make better progress. The predictors are: *(i)* the *Time-per-policy (TP) predictor*, favoring the approach more likely to classify more policies in a single execution, and *(ii)* the *Remaining-time (RT) predictor*, favoring the approach more likely to complete faster. If the predictors disagree on the approach, *Config2Spec* runs the data-plane analyzer, we explain the reason for this choice shortly.

high-level behavior The predictors dynamically identify the different stages of the algorithm. In the beginning, sampling concrete environments is likely to provide the fastest progress, as at this stage the dense policies have not been pruned yet. Therefore, the TP predictor prefers data-plane analysis initially. After most of the dense policies have been pruned, sampling environments may not significantly decrease the number of candidate policies anymore. At this point, the TP predictor starts to prefer control-plane verification. Thus, the choice is then up to the RT predictor. It determines whether *Config2Spec* switches to control-plane verification. If running data-plane analysis for the remaining concrete environments is likely to be faster than running control-plane verification on the remaining unclassified policies, the RT predictor prefers data-plane analysis. Otherwise, it prefers control-plane verification. This choice depends on the failure model: if it captures a small number of concrete environments, enumerating all of them can be faster than verifying the remaining set of candidate policies. In our running example and the failure model $L_{symbolic} = L$ and $k = 1$, this is the case. To conclude, the joint behavior of the predictors is to prefer control-plane verification whenever *(i)* there is a large number of concrete environments and *(ii)* most remaining policies are true policies (i.e., part of the specification) or sparse violations.

COMPUTATION The predictors rely on statistics of the previous runs. The TP predictor is implemented by tracking two times: $T^{TP}_{analysis}$ and T^{TP}_{verify}, which record the average time to classify a single policy through analysis or verification (respectively). For $T^{TP}_{analysis}$, this time is computed by taking the ratio of the execution time of the last run of the data-plane analysis and the number of policies which were pruned as a result of this analysis. For T^{TP}_{verify}, this time is computed similarly by taking the ratio of the execution time of the last run of the verifier and the number of policies which were classified by the verifier. The latter number is one of the following. If the verifier proved all policies hold, it equals the number of policies. Otherwise, if the verifier returned a counterexample, this number equals to the number of policies which were discovered as violations (i.e., the counterexample violated them). The TP predictor prefers the data-plane analyzer if $T^{TP}_{analysis} < T^{TP}_{verify}$.

The RT predictor is implemented by tracking two, different times: $T^{RT}_{analysis}$ and T^{RT}_{verify}, which record the execution time of a single run of the analyzer and verifier (respectively). The RT predictor prefers the data-plane analyzer if the remaining time of the analyzer, obtained by multiplying $T^{RT}_{analysis}$ with the number of non-analyzed concrete environments is smaller than the remaining time of the verifier, given by multiplying $T^{RT}_{verifier}$ with the remaining number of unclassified policies.

INITIALIZATION To initialize T^{TP}_{verify} and T^{RT}_{verify}, *Config2Spec* executes the verifier on M policy sets (in our implementation, $M = 10$). It then sets T^{RT}_{verify} to the average execution time of the verifier, and T^{TP}_{verify} to the average ratio of execution time and policies verified or pruned. The estimates $T^{TP}_{analysis}$, $T^{RT}_{analysis}$ are initially 0, to guide *Config2Spec* to begin by data-plane analysis. This captures our premise that initially data-plane analysis is likely to classify more policies (the dense violations, which are the vast majority of the policies).

WINDOWS To smoothen the behavior of the predictors, the times are averaged over the last N runs of the analyzer or verifier (in our implementation, we use $N = 10$).

4.4 DATA-PLANE ANALYSIS

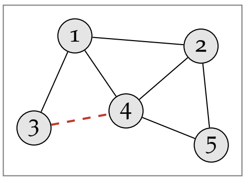
In this section, we present the key ingredients of running the data-plane analysis in *Config2Spec*: the selection of the next concrete environment to analyze (`PickCE`), the computation of the data plane for that environment (`DPCompute`) and the inference of the policies from the data plane (`InferPol`).

4.4.1 *Selection of Concrete Environments*

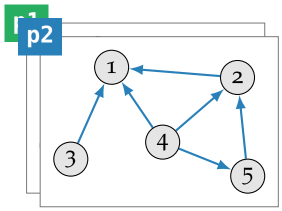
At every iteration, one concrete environment is analyzed. The choice of this environment has a great impact on the overall runtime of the system. Thus, we design a sampling technique to pick the next concrete environment to prune a large number of policies from the set of candidates (`cands`). We call this technique *policy-aware sampling* as the next environment is picked based on the *policy graph*, a concept reflecting the current set of candidate policies, which we describe next.

POLICY GRAPH The *policy graph* for a given concrete environment is a copy of the network topology, in which each link is augmented with the number of policies that forward traffic along this link. We say, for example, that a reachability policy between `r` and `p` forwards traffic along a link, if that link is part of a path in the forwarding graph of `p` from `r` to `p`. We define it similarly for the other policies. The policy graph allows us to identify the links on which large numbers of policies depend. Thus, we can pick a concrete environment in which these links are down. If the policies indeed hold only thanks to these links, they will be discovered as violations when analyzing this concrete environment.

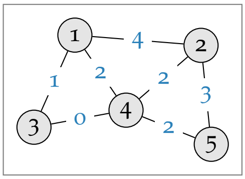
We next define the policy graph. Given a network topology, a configuration, and a concrete environment, the policy graph extends the network topology with a mapping of links to weights (integers). The weight of a link



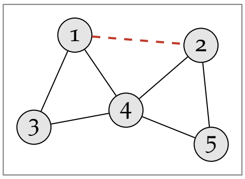
(a) Previous environment.



(b) Previous forwarding graphs.



(c) Policy graph.



(d) Next environment.

FIGURE 4.4: The policy graph is computed from the forwarding graphs of a previously analyzed concrete environment and guides us to an environment likely to prune more policies.

represents the number of unclassified policies whose traffic is forwarded along that link. The weight is computed from the *forwarding graphs* of the concrete environment.

Example 4.2. *Figure 4.4 illustrates the concept of the policy graph using our running example (Figure 4.1). Here, we are given an (already analyzed) concrete environment where all links are up, but the one between routers* `3` *and* `4` *(Figure 4.4a). In this example, there are two destinations (*`p1` *and* `p2`*) and hence two forwarding graphs (Figure 4.4b). For simplicity's sake, consider the following unclassified policies for destination* `p2`*:* `reachability(i, p2)`*, where* `i` *ranges over all five routers, and* `loadbalancing(4, p2)`*, which holds since router* `4` *has three paths to router* `2` *in the forwarding graph of* `p2`*. In this setting, the policy graph (Figure 4.4c) maps, for example, the link between* `1` *and* `3` *to* 1 *as only a single policy (*`reachability(3, p2)`*) depends on this link. Similarly, the link between* `2` *and* `5` *gets assigned a weight of* 3 *as three policies (*`reachability(4, p2)`*,* `reachability(5, p2)` *and* `loadbalancing(4, p2)`*) use this link. The link between* `1` *and* `2` *is assigned a weight of* 4 *because of the three reachability policies (*`reachability(1, p2)`*,* `reachability(3, p2)`*,* `reachability(4, p2)`*) and* `loadbalancing(4, p2)`*. The link between* `3` *and* `4` *has a weight of* 0 *as it is down and therefore is not used by any policy.*

POLICY-AWARE SAMPLING Based on the idea of the policy graph, we design a policy-aware sampler for `PickCE`. The policy-aware sampler picks the next concrete environment to analyze based on the policy graph of the previously analyzed concrete environment and the current set of unclassified policies (`cands`$\setminus$`verified`). This is done by selecting the links to add to L_{down} based on a probability distribution, which is proportional to the links' weights in the policy graph. The links' weights are computed by iterating over all unclassified policies (`cands`$\setminus$`verified`) and counting, for each link, the number of policies that are forwarded along it. The probability distribution is needed to avoid getting stuck: a deterministic approach which adds the heaviest links to L_{down} can result in an oscillation between two concrete environments which already have been analyzed (we observed this phenomenon in practice). Adding non-determinism mitigates this issue, and in case it cannot, `PickCE` resorts to returning a random concrete environment which has not yet been analyzed. In the beginning, *Config2Spec* analyzes the concrete environment in which all symbolic links are up.

Example 4.3. *For our running example and the policy graph in Figure 4.4c, it assigns the link* `1-3` *to the probability* $\frac{1}{14}$, `2-5` *to* $\frac{3}{14}$, *and* `1-2` *to* $\frac{4}{14}$. *Assuming the usual failure model (*$L_{symbolic} = L$ *and* $k = 1$*), it then picks the next concrete environment by choosing one link that is down based on the distribution. For example, it picks the link* `1-2` *(Figure 4.4d).*

4.4.2 *Analysis of a Concrete Environment*

We now explain `DPCompute` and `InferPol`, which together compute all policies that hold for a given concrete environment and configuration.

The `DPCompute` algorithm executes two steps. First, for each router in the network, it computes the router's forwarding state. The forwarding state of a router is a list of destination prefix and next hop pairs. A pair (`p`,`w`) in the forwarding state of router `r` indicates that traffic reaching `r` for destination `p` is sent to router `w`. Computing the forwarding state of the routers is not trivial, however, there are solutions to efficiently compute them (e.g., Batfish [26]).

In the second step, `DPCompute` builds from the routers' forwarding states the forwarding graphs. It builds one forwarding graph for each equivalence class of destination prefixes (i.e., prefixes which have the same forwarding graph). The forwarding graph of a prefix `p` is a directed graph in which we have a link from router `r` to `w` if, according to `r`'s forwarding state, traffic for `p` is sent to `w`.

From the forwarding graphs, `InferPol` computes the policies by leveraging graph algorithms. For `reachability` and `waypoint` policies, it builds the *dominator tree* of all forwarding graphs. A dominator tree is a tree rooted at the destination of the forwarding graph. Its nodes are all routers that have at least one path to the destination. A router `a` is a child of a router `b` if *(i)* traffic from router `a` to the destination must pass through router `b` and *(ii)* for any other router `c` such that traffic from `a` must pass through it, traffic from `b` must also pass through it. `InferPol` infers a `reachability(r, p)` policy for every node `r` in the dominator tree of `p`. It further infers `waypoint(r,w,p)` for all routers `r` which are dominated by a waypoint `w` in the dominator tree of `p`. For `loadbalancing`, it computes the shortest paths in the network and infers `loadbalancing(r,p)` for routers `r` with multiple paths of the same cost available to reach destination `p`. For `isolation`, it infers `isolation(r,p)` for every router `r` and prefix `p` for which it has not inferred `reachability(r,p)`.

4.5 CONTROL-PLANE VERIFICATION

In this section, we present the two ingredients of the control-plane verification in *Config2Spec*: the selection of the next policies to verify (`PickPolicies`) and their verification (`CPVerification`).

CPVERIFICATION We begin with `CPVerification`, which takes as input a set of policies, the network configuration and the failure model. It checks whether all policies hold for any concrete environment meeting the failure model (for the given network configuration), or returns a counterexample.

Technically, the verifier symbolically encodes the configuration and the failure model as logical constraints: φ_{net} and φ_{fmodel}. The set of policies is encoded as a conjunction over formulas encoding the policies: $\varphi_{pols} = \bigwedge_{pl \in pols} \varphi_{pl}$. The verifier checks the satisfiability of $\varphi_{net} \land \varphi_{fmodel} \land \neg\varphi_{pols}$. If it is unsatisfiable, then all policies in *pols* hold. If the formula is satisfiable, then there is a counterexample, i.e., a concrete environment captured by the failure model, which under the given configuration violates φ_{pols} (i.e., at least one policy is violated). While the challenge of verifying network policies is not trivial, there are effective solutions (e.g., Minesweeper [16]).

PICKPOLICIES This procedure takes the set of candidate policies (`cands`) and verified policies (`verified`) and returns the next set of policies to verify (from `cands` $\setminus$ `verified`). Since verifying is computationally expensive, the goal is to minimize the overall execution time of the verifier. By choosing a set of policies which have a dependency, the overall execution time of verifying them can be smaller than if they were verified one by one. Towards this goal, `PickPolicies` returns a maximal set of policies with the same destination prefix `p`.

We pick `p` arbitrarily, as once *Config2Spec* chooses to run the verifier, usually most policies are true policies.

Our grouping approach is always at least as good as verifying the policies one by one. The reason is that with each query to the verifier, at least one policy is classified. In the worst case, only one policy is classified as violation (if the verifier returned a counterexample which satisfies all policies but one). In a better case, several policies are classified as violation. In either of these cases, the violated policies are removed from `cands`, while the other policies in the set remain in `cands` (and will be verified in a later execution of `CPVerfication`). In the best case, all policies are classified as true policies. Namely, we can only gain from verifying multiple policies in the same execution of the verifier. Further, our grouping is maximal – grouping of policies with different prefixes is not helpful, as each prefix has a different forwarding graph, and so the verifier does not gain from grouping such policies.

4.6 TOPOLOGY-BASED TRIMMING

In this section, we describe `TopoTrim`, a technique which reduces the load on the control-plane verification by analyzing the failure model and the network topology. `TopoTrim` classifies policies as violations if their minimal connectivity requirements are not met under the given failure model.

`TopoTrim` is executed the first time *Config2Spec* chooses to run the verifier. It relies on the insight that some policies can be classified as violations directly from the network topology and failure model. For example, consider the network in Figure 4.1 and the failure model with $L_{symbolic} = L$ and $k = 2$ (i.e., up to two link failures). We can infer that `reachability(3, p1)` cannot hold as `3` can become disconnected from the rest of the network if both links connected to it fail. For the same reason, any `waypoint` or `loadbalancing` policy where `3` is involved can be classified as violation.

To prune such policies, `TopoTrim` computes the $(k+1)$-edge-connected components of the topology for a failure model with k permitted failures. A $(k+1)$-edge-connected component is a set of nodes which remain connected even after removing any k edges. For example, for the network in Figure 4.1 and the same failure model (where $k = 2$), the following routers are in a 3-edge-connected component: $\{\texttt{1}, \texttt{2}, \texttt{4}\}$.

There are efficient algorithms to compute $(k+1)$-edge-connected components, however they do not support links that must be up or down (L_{up} or L_{down}). To take these into account, `TopoTrim` first removes from the topology all links in L_{down}, updates k to $k - |L_{down}|$, and then, for each link in L_{up}, it adds k additional links between the routers to simulate that these routers are $(k+1)$-edge-connected. For example, for $L_{up} = \{(\texttt{1},\texttt{3})\}$, $L_{down} = \emptyset$ and $k = 2$, it adds two more edges between `1` and `3`, so they are considered 3-edge-connected.

Based on this, `TopoTrim` classifies the following policies as violations (which are thus removed from `cands`). The policies `reachability(r,p)` and `loadbalancing(r,p)`, for any router `r` and prefix `p` such that $(\texttt{r}, \texttt{r}_\texttt{p})$ is not in a $(k+1)$-edge-component, where $\texttt{r}_\texttt{p}$ is the router attached to `p`. The policy `waypoint(r,w,p)` is classified as violation for any routers `r` and `w` and a prefix `p` such that *(i)* $(\texttt{r},\texttt{w})$ is not in a $(k+1)$-edge-component or *(ii)* $(\texttt{w}, \texttt{r}_\texttt{p})$ is not in a $(k+1)$-edge-component, where $\texttt{r}_\texttt{p}$ is the router attached to `p`.

4.7 EXPERIMENTAL EVALUATION

In this section, we evaluate *Config2Spec* on multiple topologies to address the following research questions:

RQ1 How does *Config2Spec* scale to realistic topologies? We show that even for large networks with 158 routers and 189 links, it completes within 2.7 hours for OSPF configurations and 13.7 hours for BGP configurations (Section 4.7.1).

RQ2 How does *Config2Spec* compare to the baselines? We show it improves the best one by up to a factor of 8.3 (Section 4.7.2).

RQ3 How do the domain-specific techniques contribute to *Config2Spec*? We show that *(i)* the Policy-Aware sampler leads to smaller candidate sets by up to a factor of 2 compared to a random sampler, and obtains them with fewer samples, and *(ii)* topology-based trimming and policy grouping reduce the queries by up to a factor of 2 500 (Section 4.7.3).

RQ4 Can *Config2Spec* be run on a real network configuration? We illustrate this on the Internet2 configuration (Section 4.7.4).

IMPLEMENTATION *Config2Spec* is implemented in 5*k* lines of Python and Java code.[1] It computes the routers' forwarding states (Section 4.4.2) using Batfish [26], and verifies policies using Minesweeper [16]. We extended Minesweeper with the `waypoint` and `loadbalancing` policies. We note that while our implementation supports only configurations and features supported by these two third-party tools, our approach is not limited to specific configuration types or features.

Config2Spec takes as input the routers' configurations and a failure model. It outputs all policies that hold for the provided input. For large networks, we assume the network operator provides a list of devices that act as waypoints (e.g., middleboxes). In our experiments, we simulate it by randomly picking 20% of the routers to serve as waypoints.

EXPERIMENT SETUP To study how *Config2Spec* scales as a function of the topology size, we picked three topologies (small, medium, and large) from the Topology Zoo collection [47]: BICS with 33 routers connected by 48 links, Columbus with 70 routers and 85 links, and US Carrier with 158 routers and 189 links. We used NetComplete [23] to synthesize OSPF and BGP configurations using its path-ordering specifications for 2, 4, 8 and 16 prefixes. For each configuration type and topology, we generated 5 configuration sets.

For each set of router configurations, *Config2Spec* computes all policies which hold, for all four policy types in Table 4.1. We consider three failure models, where k is 1, 2, or 3, and we fix $L_{up} = L_{down} = \emptyset$ and $L_{symbolic} = L$ (i.e., any link can be up or down). The reported results are averaged over these runs and the two configuration types (i.e., OSPF and BGP). We ran all experiments in virtual machines with 32 GB of RAM and 12 virtual cores running at 2.3 GHz.

4.7.1 *Scalability of* Config2Spec

We begin by studying how *Config2Spec* scales to realistic topologies. To this end, we ran experiments on all three topologies and three failure mod-

1 Code is available at https://github.com/nsg-ethz/config2spec.

Topology	k	Config	Overall	DPA	CPV
BICS	1	OSPF	38.8 s	100%	0%
		BGP	68.3 s	100%	0%
	2	OSPF	228.8 s	30%	70%
		BGP	1 341.2 s	85%	15%
	3	OSPF	117.4 s	27%	73%
		BGP	319.7 s	14%	86%
Columbus	1	OSPF	398.0 s	100%	0%
		BGP	457.2 s	100%	0%
	2	OSPF	1 328.1 s	18%	82%
		BGP	6 772.0 s	17%	83%
	3	OSPF	907.0 s	27%	73%
		BGP	2 074.1 s	18%	82%
US Carrier	1	OSPF	6 386.2 s	100%	0%
		BGP	6 813.4 s	100%	0%
	2	OSPF	10 528.4 s	15%	85%
		BGP	49 151.0 s	6%	94%
	3	OSPF	2 542.5 s	59%	41%
		BGP	5 873.3 s	34%	66%

TABLE 4.2: Execution time of *Config2Spec* as a function of the network topology, number of failures and configuration type.

els, and measured the time *Config2Spec* spent on the data-plane analysis part – including `PickCE`, `DPCompute` (which invoked Batfish), and `InferPol` – and the time spent on the control-plane verification part – including `PickPolicies` and `CPVerification` (which invoked Minesweeper). We ignored the other parts as they completed in negligible times (e.g., `TopoTrim` completed in five seconds for US Carrier and less than a second for BICS).

Table 4.2 shows the overall execution time (*Overall*) and how it is split between data-plane analysis (*DPA*) and control-plane verification (*CPV*) as a function of the topology, the number of failures (k), and the configuration type (*Config*). For example, for the US Carrier topology with $k = 3$ and

Topology	k	Candidates	Specification	Percent
BICS	1	2 526.9	1 008.1	40%
	2	2 504.4	304.0	12%
	3	2 482.1	57.6	2%
Columbus	1	13 290.2	4 517.1	34%
	2	13 150.4	350.4	3%
	3	13 271.0	27.2	0.2%
US Carrier	1	93 416.2	17 908.3	18%
	2	85 021.0	702.8	0.8%
	3	98 837.6	6.8	0.01%

TABLE 4.3: The initial number of candidate policies and the final number of policies that are part of the specification which *Config2Spec* returns. Percent shows the fraction of the policies of all candidate policies.

OSPF configurations, *Config2Spec* completed within 43 minutes, where 59% of that time was spent on data-plane analysis.

The results show that even for the US Carrier topology with its 158 routers and 189 links, *Config2Spec* mined the specification in a reasonable time (within 2.7 hours, for OSPF, and 13.7 hours, for BGP). The results also demonstrate that the runtime mainly depends on the network size, secondly on the failure model, and lastly on the configuration type. This is expected: the larger the network, the larger the set of candidate policies and the set of concrete environments (whose size also depends on the failure model). In contrast to the effect of the network size on the execution times, the permissiveness of the failure model shows a different trend: execution times increase from $k = 1$ and $k = 2$, but drop for $k = 3$. This is thanks to the topology-based trimming (Section 4.6), which becomes very significant for $k = 3$ (or higher values of k). For the evaluated topologies, most router pairs are not 4-edge-connected, thus many policies are pruned. We provide more details on trimming in Section 4.7.3. The results show also that for $k = 1$, *Config2Spec* only performs data-plane analysis. This is because the number of concrete environments is significantly smaller than the number of candidate policies throughout the execution, leading the RT predictor to favor data-plane analysis. Finally, the results show that for BGP configurations, the execution time is higher than for OSPF configurations. This is mainly due to Minesweeper, for which we observe a five to ten times increase in the verification time for BGP compared to OSPF.

Table 4.3 reports the number of candidate policies and the number of policies in the specification, for each topology and failure model, averaged across the different configuration sets and configuration types. The reported number of candidate policies is the number of policies that hold for the first concrete environment (*Config2Spec* always begins with data-plane analysis). We consider this set as the initial set of candidates, rather than all instantiations of the four policy types (Table 4.1), as the latter contains many policies which no concrete environment satisfies.

The results indicate that as the network size increases, the number of candidate policies increases, while the specification size (i.e., the number of policies that hold for all concrete environments) significantly drops. This demonstrates the challenge of *Config2Spec* to search in the large space of candidate policies for the small set of policies that hold.

4.7.2 *Comparison to Baselines*

We compare *Config2Spec* to the two baselines in Section 4.1: *(i)* a data-plane analysis approach, which enumerates all data planes to infer the specification, and *(ii)* a control-plane verification approach, which verifies the candidate policies one by one. As neither of the baselines scales to the larger networks considered in the last section, in this experiment, we use three grid topologies of sizes: 4 by 5, 5 by 5 and 6 by 5. We generated five sets of OSPF configurations per topology and used the failure model $L_{symbolic} = L$ with k ranging from 1 to 3.

Figure 4.5 shows the execution time of each approach as a function of the topology and failure model. For $k = 2$ and $k = 3$, *Config2Spec* outperforms both baselines: the data-plane analysis by 10.2x on average and up to 41.0x, and the control-plane verification by 3.8x on average and up to 8.3x. For $k = 1$, data-plane analysis is faster than *Config2Spec* because of *Config2Spec*'s initialization time (i.e., in the beginning, *Config2Spec* verifies a few policies when initializing the predictors' times, see Section 4.3). Still, the overhead of *Config2Spec* is small (data-plane analysis was faster on average by 24 seconds and by up to 37 seconds).

The results also show that both baselines have benefits. For less permissive failure models, data-plane analysis performs better than control-

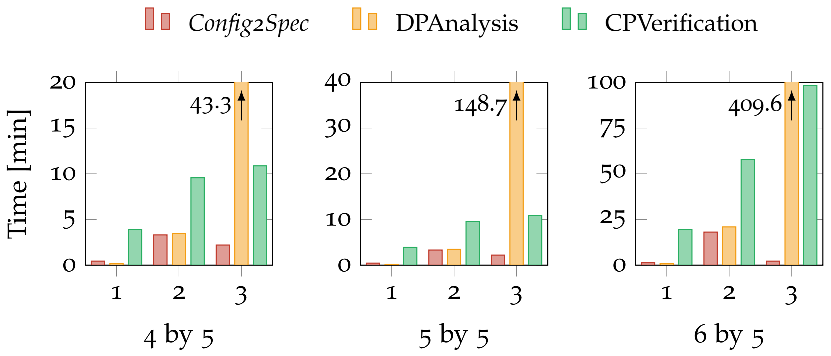


FIGURE 4.5: *Config2Spec* compared to the baselines of data-plane analysis and control-plane verification on grid topologies and different failure models. The bars of DPAnalysis and $k = 3$ are cut, and their maximum value is denoted next to them.

plane verification, whereas for permissive failure models it is the other way around. This demonstrates the advantage of the dynamic combination of *Config2Spec*.

4.7.3 *Domain-specific Techniques*

We next study how the domain-specific techniques improve *Config2Spec*'s performance. We study the following aspects: *(i)* how the Policy-Aware sampler (Section 4.4.1) helps reducing the number of concrete environments *Config2Spec* analyzes, and *(ii)* how topology-based trimming (Section 4.6) and policy grouping (Section 4.5) decrease the number of queries posed to the verifier.

Policy-aware Sampler

We compare the policy-aware sampler (called Policy-Aware) to a baseline which randomly picks a new concrete environment (called Random). We compare them by instantiating `PickCE` with each approach and running *Config2Spec* on the Topology Zoo topologies with the failure model $L_{symbolic} = L$ and $k = 3$, and with five sets of OSPF configurations and five sets of BGP configurations.

	Policy-Aware		Random	
Topology	Samples	Candidates	Samples	Candidates
BICS	36.4	36.5%	42.1	39.5%
Columbus	71.0	16.6%	79.0	26.4%
US Carrier	113.8	9.6%	122.1	18.6%

TABLE 4.4: Comparison of the number of samples and the remaining candidate policies before *Config2Spec* switched to the verifier when using a Policy-Aware sampler and a random baseline.

Table 4.4 shows for each approach, the number of concrete environments which were analyzed before *Config2Spec* transitioned to the verifier, and it shows the percentage of policies that remained in the candidate set left to verify (i.e., the percentage of remaining policies out of the policies that hold for the first sample). For example, for BICS, Policy-Aware sampling required on average 36.4 samples before *Config2Spec* switched to verification, and at this point the size of the candidate policy set was reduced to 36.5% of the initial policy set (i.e., the set of policies which hold for the first sample).

Generally, the smaller the set of remaining policies (i.e., the closer the candidate set to the network specification is), the better. As a secondary goal, the number of analyzed concrete environments should be relatively small. The results indicate that Policy-Aware sampling always obtains a better reduction in the size of the candidate set compared to Random sampling. They also show that on average Policy-Aware sampling typically required fewer samples than Random sampling. However, we note that in 6 out of the 30 experiments, Random sampling switched to verification before Policy-Aware sampling did. This is not because Random sampling made better progress. In contrary, the TP predictor decided to switch, as it observed that the concrete environments picked by Random sampling were not effectively pruning policies anymore.

Table 4.5 shows. for each experiment, the relative size of the candidate sets for both approaches when *Config2Spec* with Policy-Aware sampling transitioned to verification. For example, in one experiment using BICS, Policy-Aware sampling transitioned to verification after 32 samples, and at that point the number of candidate policies was 970, while for Random sampling, after 32 samples, there were 1 124 candidate policies, making

Topology	Candidates Ratio	Additional Samples
BICS	89.7%	45.7
Columbus	60.5%	109.0
US Carrier	51.6%	> 500.0

Table 4.5: The ratio between the candidate sets of the two sampling approaches when Policy-Aware sampling transitions to verification and the number of additional samples required for Random sampling to achieve the same reduction in the candidate set.

	Policy-Aware		Random	
Topology	PickCE	DPAnalysis	PickCE	DPAnalysis
BICS	22.1 ms	1.4 s	0.5 ms	1.3 s
Columbus	63.1 ms	8.3 s	0.7 ms	7.7 s
US Carrier	358.2 ms	57.8 s	1.4 ms	51.6 s

Table 4.6: Comparison of the runtime with Policy-Aware sampling and Random sampling.

the ratio 86.3%. In Table 4.5, Candidates Ratio shows the average over the ten runs. We also checked how many additional samples Random sampling required to reduce the candidate policies to (at most) the size obtained with Policy-Aware sampling. For example, in that experiment for BICS, Policy-Aware sampling required 32 samples to reduce the candidates to 970 policies, while Random sampling required 43. Hence, Random sampling needed 11 additional samples. In Table 4.5, Additional Samples shows the average of this number. The results indicate that Policy-Aware sampling not only obtains a smaller candidate set, but reaches it significantly faster.

Table 4.6 shows the execution times: PickCE shows the execution time of the sampler, while DPAnalysis shows the overall execution time of a single data-plane analysis (i.e., `DPCompute` and `InferPol`). The results show that while Policy-Aware sampling takes (as expected) more time than Random sampling, the overhead is negligible compared to the overall execution time of the data-plane analysis.

Topology-based Trimming and Policy Grouping

We next evaluate the topology-based trimming and policy grouping in reducing the number of queries to the verifier. We ran the experiments for the three topologies and the failure model with $k = 2$ and $k = 3$ (for $k = 1$, *Config2Spec* only performs data-plane analysis as explained in Section 4.7.1). We measured how many queries to the verifier each technique saved. In every experiment, we recorded the number of policies *Config2Spec* had the first time it transitioned to the verification. This number, denoted B (for baseline), provides the number of queries to the verifier if we did not use either technique. We also recorded how many policies were pruned thanks to topology-based trimming. We count each policy that has been pruned as one saved query for the verifier, and denote the overall saved queries by T (for trimming). Also, we recorded how many queries were posed to the verifier (when employing policy grouping), and denote the number of queries by G (for grouping).

Figure 4.6 shows the percentage of remaining queries after each optimization: $\frac{B-T}{B}\%$ for trimming and $\frac{G}{B}\%$ for policy grouping. For example, for BICS and $k = 2$, trimming pruned 51.1% of the policies. Policy grouping saved 41.5% and reduced the overall queries to the verifier to 9.6%. Overall, the reduction was 90.3%. The results show that the combination of trimming and policy grouping can reduce the number of queries to as little as 0.04%. Trimming is especially powerful for the larger topologies and for more permissive failure models ($k = 3$). The policy grouping also significantly reduces the number of queries to the verifier. The best case is for the largest network, where trimming reduced the number of queries to 1.15% and then policy grouping reduced it to 0.04%, compared to the baseline.

4.7.4 *Running* Config2Spec *on Internet2*

Finally, we demonstrate that *Config2Spec* can handle real configurations. For this, we took a publicly available configuration of the Internet2 network from May 2015 [67]. For Batfish to be able to parse this configuration, we had to remove multiple lines from it. Mostly, these lines concerned logging statements (e.g., `system dump-on-panic;`), left-overs from the anonymization (e.g., `Firewall Stanza Removed`) and other (for our purposes) irrelevant parts (e.g., `bfd-liveness-detection no-adaptation;`). For Minesweeper to

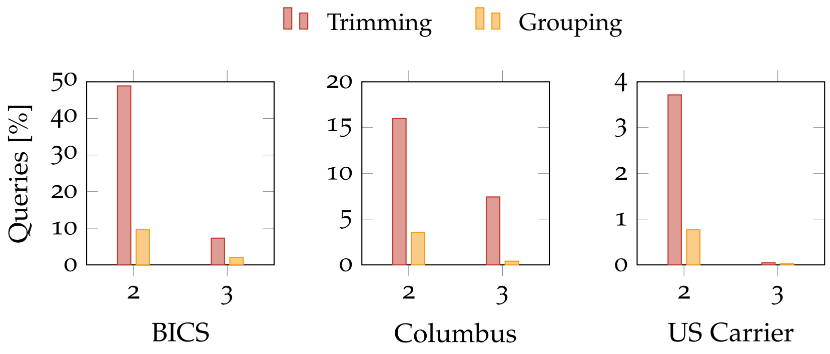


FIGURE 4.6: Reduction in the number of queries to the verifier thanks to topology-based trimming and policy grouping.

be able to verify our queries, we had to remove parts of the BGP route-maps (community-matches and empty prefix-list matches). This does not affect the output, as we only mine the specification for internal prefixes, since no external peers are connected. In total, we had more than $90k$ lines of configuration. The topology consisted of 10 routers and 18 links. For a failure model with $L_{symbolic} = L$ and k from 1 to 3, *Config2Spec* required 32, 314, and 1 805 seconds to infer the specification. It consisted of 3 962, 3 405, and 3 339 policies. The high number of policies, even for $k = 3$, stems from the fact that the five routers on the East Coast form a clique.

4.8 RELATED WORK

In this section, we survey related work across three dimensions: specification mining (in general and in the context of computer networks), network specification languages, and counterexample-guided inductive synthesis.

SPECIFICATION MINING Akin to software specifications, formal specifications are hard to write (as hard as writing the program in the first place [68]), debug, and modify [69, 70].

Our work is inspired by works on specification mining [71], where high-level specifications are automatically inferred from low-level execution of programs. One example is Daikon [72], which dynamically detects program invariants (e.g., $x \neq 0$) by running the program and observing the values the program computes.

Several previous works [73, 74, 75] have looked into mining a network's specification by observing the content of the data plane. These works are mostly limited to reachability policies, and unlike *Config2Spec*, they either approximate the specification or do not consider the impact of failures on the specification. Concretely, they only produce the network's policies which hold when all links and routers are up. In contrast, *Config2Spec* is able to mine precise network specifications for a given failure model.

Xie et al. [73] show how to compute the reachability specification for a given failure model based on the network's configuration. They compute the reachability upper bound – all policies that hold for at least one concrete environment – and lower bound – all policies that hold for all concrete environments. To scale, only an approximation of the bounds is computed. In contrast, *Config2Spec* computes the exact lower bound of reachability, as well as other policies, and thereby obtains a precise specification. Benson et al. [74] show how to mine reachability *policy units*, a high-level abstraction of pair-wise reachability, from network configurations for a single concrete environment. Like *Config2Spec*, it relies on data-plane analysis. However, unlike *Config2Spec*, failure models are not supported. Other works [75, 76] assess the management complexity of the network and its overall health, i.e., the frequency of performance and availability problems, by analyzing its configurations.

Wang et al. [77] introduce a comparative synthesis framework to learn high-level network design objectives (e.g., minimize convergence time). The operators are repeatedly presented with different scenarios, which they have to rank. Then, based on the preferences, the synthesizer is able to learn the operators' network-design "objective" function.

Anime [66] is a tool that infers a network's intent (specification) directly from its forwarding state. Anime leverages hierarchies among the features in the network (e.g., endpoints) to summarize the observed paths and provide intents of fixed size. This summarization allows it to include unobserved behavior in its intents. Thus, while Anime learns a specification that fits within a given size, *Config2Spec* learns the exact specification that the configuration enforces.

NETWORK SPECIFICATIONS Many works introduce different network specification languages, varying in their expressiveness. For example, some allow to capture traffic classes at the path-level [15, 20, 78], while others use a higher-level abstraction describing traffic classes and high-level policies such as reachability and waypointing [21]. Despite the differences, *Con-*

fig2Spec's output can be used by other tools, such as NetKAT [78], whose language can accommodate the policies we consider.

COUNTEREXAMPLE-GUIDED INDUCTIVE SYNTHESIS (CEGIS) CEGIS is a technique in program synthesis in which examples guide the search for the target program [79, 80]. Technically, from an initial set of examples (which may be empty), the synthesizer proposes a candidate program consistent with the examples and introduces it to a validator. The validator either confirms the candidate is the target program or returns a counterexample. The counterexample is added to the set of examples, guiding the synthesizer to look for a different candidate. *Config2Spec* can be seen as a synthesizer looking for (all) policies that hold for a given network configuration and failure model. Like CEGIS, it is guided by examples (the data planes) and a validator (the verifier). However, unlike CEGIS, *Config2Spec* looks for *all* valid policies (and not a single one). This poses a greater challenge, both in terms of the search space and the burden on the validator. To cope, *Config2Spec* cleverly samples examples to prune the search space (without the help of the validator), trims and groups policies to save queries to the validator and dynamically switches between sampling and verifying to expedite the search.

4.9 CONCLUSION

In this chapter, we introduced *Config2Spec*, a system to help network operators better understand their configurations and, at the same time, speed up the adoption of network validation and configuration synthesis tools. Based on the network's configuration and a failure model, *Config2Spec* automatically mines the corresponding network specification. The key insight is to dynamically switch between data-plane analysis and control-plane verification. To scale further, we integrated three domain-specific techniques: *(i)* policy-aware sampling to pick concrete environments which are more promising for policy pruning, *(ii)* policy grouping to group queries and thereby reduce verification overhead, and *(iii)* topology-based trimming to prune policies, which are infeasible for the given topology and failure model. We evaluated *Config2Spec* on different topologies and against two baselines. The results show that *Config2Spec* scales to large networks, unlike the baselines, and that our domain-specific techniques significantly contribute to the scalability.

5 UNDERSTANDING THE CAPABILITIES OF NETWORK VALIDATORS

In this chapter, we introduce *Metha*, a system which helps network operators understand the capabilities of their network validation tools and helps their developers make them more accurate.

Network validation tools can be of great help to network operators as long as they faithfully capture the network's behavior. As with any complex software, though, these tools can (and often do) have bugs that compromise their accuracy. This is not surprising: building an accurate and faithful network analysis tool is extremely difficult. Among others, one not only has to precisely capture all the different protocols' behaviors but also all of the quirks of their specific implementations. Unfortunately, every vendor, every OS, and every device can exhibit slightly different behaviors under certain conditions. Hence, it is not a question of whether these tools faithfully model the network's behavior, but rather a question of when network operators can rely on the analyses of these tools and when they cannot. *Metha* helps network operators understand exactly that: it systematically tests network validation tools and automatically finds inaccuracies in their underlying network models. *Metha* is useful for users and developers of these tools alike. For every bug it finds, *Metha* identifies the involved configuration statements and provides a minimal configuration example that can be used to reproduce the bug. *Metha* relies on black-box differential testing: it generates input configurations and compares the output of the tool under test with the output produced by the actual router software.

The main challenge behind *Metha* lies in efficiently and thoroughly covering the gigantic search space of all possible configurations. On the one hand, there are hundreds of configuration statements, each of which can take many possible parameters. And yet, as our analysis reveals, most of the bugs only manifest themselves when specific configuration statements and values are present. On the other hand, not all configurations are useful for testing. One needs to make sure to only generate syntactically- and semantically-valid configurations that allow for meaningful control-plane computations and fully exercise the network model. *Metha* addresses this challenge by first reducing the search space through restricting the parameters to their boundary values and then phrasing the search as a combinatorial testing problem. To ensure syntactically- and semantically-valid configurations, *Metha* relies on a hierarchical grammar-based approach.

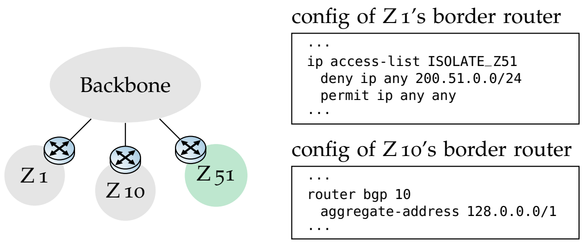
Next, we summarize our main contributions in this chapter:

- We present a testing system capable of finding bugs in the network models of state-of-the-art network validation tools (Section 5.2);
- we present a precise localization procedure relying on delta debugging to isolate bugs and the configuration statements causing them (Section 5.5);
- and we provide an end-to-end implementation of *Metha* and show that it finds real (and unknown) bugs in all the tested tools (Section 5.7).

This chapter is organized as follows. Section 5.1 demonstrates the consequences of inaccuracies in a network model on the example of a state-of-the-art network validator. Section 5.2 provides an overview of the entire system. Section 5.3 defines the space of all possible network configurations. Section 5.4 presents *Metha*'s approach to thoroughly cover the search space. Section 5.5 shows iterative delta debugging to isolate the bug-inducing configuration statements. Section 5.6 highlights the key points of *Metha*'s implementation. Section 5.7 evaluates *Metha* on state-of-the-art network validation tools. Section 5.8 discusses *Metha*'s testbed and the scope of the tests. Section 5.9 reviews related work in the area. Finally, Section 5.10 concludes the chapter.

5.1 MOTIVATION

We now illustrate how subtle bugs in the network model of network validators can lead operators to deploy erroneous configuration changes. We start with two case studies on common configuration features known for easily causing forwarding anomalies: route aggregation and redistribution. In these situations, validating the change with a network validation tool is of utmost importance, provided the analysis is correct. Finally, we end with a collection of Cisco IOS configuration statements whose semantics were not correctly captured by Batfish [81]. All the bugs we report in this section were discovered by *Metha*.



config of Z 1's border router

```
...
ip access-list ISOLATE_Z51
  deny ip any 200.51.0.0/24
  permit ip any any
...
```

config of Z 10's border router

```
...
router bgp 10
  aggregate-address 128.0.0.0/1
...
```

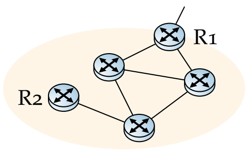
FIGURE 5.1: Zone 51 has to be isolated from all the other zones. This is achieved through access-lists at the border routers with the exception of zone 10 where it was forgotten.

5.1.1 *Example 1: Excess Null Route*

Consider the network in Figure 5.1. It consists of a backbone with multiple zones attached to it. The backbone and the zones are interconnected using BGP. Each zone receives a default-route from the backbone. Zone 51 hosts critical infrastructure in the prefix `200.51.0.0/24`, which should not be accessible from any other zone. To enforce this, the routers connecting the zones to the backbone have an access-list (ACL) in place to filter that traffic. However, in zone 10, this ACL was forgotten and, instead, there happens to be a left-over statement from a previous configuration: "`aggregate-address 128.0.0.0 128.0.0.0`". This statement directs the router to advertise the specified aggregate route *if* any more-specific BGP routes in that range exist in the routing table.

PROPERTY VIOLATION Due to the lack of an ACL on the border router, the requirement that mandates to keep zone 51 isolated is violated: traffic from zone 10 can reach zone 51.

ANALYZER MISHAP When used on the network above, Batfish, a recent network validation tool, will falsely assert that zone 51 is fully isolated. The problem is due to the semantics of the left-over `aggregate-address` statement. Batfish wrongly activates the aggregate because of a non-BGP route in the routing table and installs the corresponding `null` route. Because of this `null` route, Batfish wrongly assumes that all traffic in zone 10 falling within the aggregate range will be dropped. In practice, the routers only install a `null` route if a BGP route within the aggregate is present, which is not the case here.



config of Z10's border router

```
...
ip route 200.0.0.0/20 Null
...
router ospf 1
  redistribute static
...
```

FIGURE 5.2: All routers should be able to reach the Internet. The static route at R2 creates a blackhole and violates that.

5.1.2 *Example 2: Incomplete Redistribution*

Consider the small company network depicted in Figure 5.2. It consists of a single OSPF area. R1 acts as Internet gateway and announces a default-route internally. A static route on R2 drops all the traffic for `200.0.0.0/20` by directing it to the `null` interface. This is intended. What is not intended, however, is the `redistribute static` command at R2.

PROPERTY VIOLATION The following reachability property must always hold: all routers, with the exception of R2, are able to reach the entire Internet. However, this property is violated since R2 redistributes the static route in the network and, in turn, creates a blackhole for `200.0.0.0/20`.

ANALYZER MISHAP When run on this network, Batfish will falsely attest that all routers, with the exception of R2 can reach the entire Internet. The problem is the redistribution command. By default, Cisco routers only redistribute classful networks [82] and only by specifying the `subnets` keyword, they also redistribute any subnets of them. Less-specific networks, however, are always redistributed regardless of the `subnets` keyword (e.g., `200.0.0.0/20` is less specific than the corresponding class C network `200.0.0.0/24`). Batfish's network model does not incorporate that as it only redistributes classful networks and not less-specific networks.

5.1.3 *Selection of Bugs*

In addition to the two bugs illustrated in the previous examples, we found several other configuration statements that trigger bugs. We present a selection of them in Table 5.1 alongside a short description of the observed behavior and possible consequences. All of the presented bugs concern Cisco IOS configuration statements. In our tests, we also used Juniper configurations and found that, in many cases, the same bugs occur. Hence, some of these bugs are not due to vendor-specific behaviors but due to general inaccuracies of the network model.

Feature	Description	Possible Consequence
`max-metric router-lsa`	The model sets maximum metric not only for point-to-point links, but also for stub links. This should only be done when the keyword `include-stub` is used.	A router might appear to be free of traffic and safe to reboot, even though it is not.
`default-information originate`	The OSPF routing process should only generate a default-route if the route table has a default-route from another protocol. The model, however, also announces a default-route if there is one in the routing table of different OSPF type, i.e., E1 type.	Additional default-routes might appear in the routing tables.
`distance XX`	The model does not consider any changes to the administrative distance.	The forwarding state could be completely wrong.
`area X range A.B.C.D/Y`	When summarizing routes between OSPF areas, the model does not insert a null route for the summary to prevent routing loops.	A routing loop could be falsely detected.
`set community no-export`	When redistributing a static route into BGP and setting the `no-export` community, the model still advertises the route to its eBGP neighbors.	Reachability properties could be falsely asserted.
`neighbor A.B.C.D maximum-prefix X`	Even when a BGP neighbor advertises more prefixes than the specified threshold, the model does not drop the peering to the neighbor.	Reachability properties could be falsely asserted.

Table 5.1: A selection of Cisco IOS configuration features that are incorrectly modeled by Batfish [81] as found by *Metha*.

5.2 OVERVIEW

In this section, we first present the key insights enabling *Metha* to efficiently uncover bugs in network analyzers. Then, we provide a high-level overview of *Metha*.

5.2.1 *Key Insights*

The main challenge in testing network analyzers is that bugs may occur rarely and only for very specific configurations, which we address with a combination of five insights:

GENERATING VALID INPUTS USING A GRAMMAR-BASED APPROACH When testing network analyzers, it is crucial to use syntactically- and semantically-valid configurations, meaning the configurations need to be parseable and constraints have to be met such that actual computation takes place in the network. Our key insight is to use a hierarchical grammar-based approach. Approaching it in multiple steps allows to first build a basic structure by resolving the intra- and inter-device constraints, ensuring semantical validity. Then, this basic structure is completed using grammar-based configuration generation ensuring, syntactical validity.

REDUCING THE SEARCH SPACE THROUGH BOUNDARY VALUES The search space of all possible configurations is prohibitively large. Even a single parameter, such as an OSPF cost, for example, already has 2^{16} possible values to test. By focusing the testing on the boundary values (the minimum, maximum, and a "normal" value in between), we are able to reduce the search space significantly.

EXPLORING THE SPACE WITH COMBINATORIAL TESTING Network devices support a wide variety of configuration features that all need to be tested not just by themselves but also their interactions. Hence, we use combinatorial testing to design a test suite that systematically covers all pairwise interactions of configuration features.

COMPARING THE TESTED TOOL'S OUTPUT TO GROUND TRUTH Detecting crash bugs is straightforward as the tool will just fail or report an error. Silent bugs, on the other hand, can only be detected by comparing the output to a ground truth, which is hard to come by. We address this by leveraging a testbed running real router images as an oracle.

ISOLATING BUGS WITH DELTA DEBUGGING Finally, once one identifies a network configuration that triggers a bug, one needs to identify the exact configuration statements causing it, to provide any useful insights to the tool's developer. Therefore, we use iterative delta debugging to obtain a minimal configuration example, which reproduces the bug and can even be used as a future test case.

5.2.2 Metha

Metha operates in two phases as shown in Figure 5.3: First, it aims to find network configurations exhibiting discrepancies between the tool under test and the oracle. To that end, the test coordination determines all the tests that should be run. Second, it identifies the configuration statements responsible for the observed discrepancies through fault localization.

Input and Output

Metha takes two inputs: *(i)* a physical topology, i.e., an undirected graph; and *(ii)* a set of configuration features to be tested, such as, route-maps, and route-summarization. For every discovered bug, *Metha* creates a report, which consists of the identified discrepancy between the routing tables of the tool and those of the oracle, the configuration statements causing it and a configuration set to reproduce it.

Phase I: Test Coordination (Sections 5.3 and 5.4)

The configuration features and the topology provided as input define the search space of *Metha*'s testing efforts which consists of all possible configurations that can be built using these features.

This search space of network configurations is prohibitively large. Therefore, *Metha* first reduces the values of all parameters to their boundary values, which means it only uses the two extreme values (i.e., the minimum and the maximum) and one "normal" value. Even with this reduction, it is difficult to systematically cover the entire search space. Hence, *Metha* creates a test suite relying on combinatorial testing, which allows it to cover all pairs of feature and parameter combinations while requiring a minimal number of tests. Each test in the test suite consists of a set of configuration statements that should be active.

Phase I: Testbed (Section 5.6)

For every single test, *Metha* generates the device configurations based on the statements that are provided by the test suite. Then, it runs these configurations in the tool and the oracle. Once both have converged, *Metha* analyzes the routing tables of the two tools and reports any discrepancies.

Phase II: Fault Localization (Section 5.5)

A discovered discrepancy can be caused by multiple bugs in the network analyzer. Therefore, *Metha* applies delta debugging to identify every single bug and the configuration statements causing it. It does so by iteratively testing subsets of the active configuration statements until the entire discrepancy is resolved.

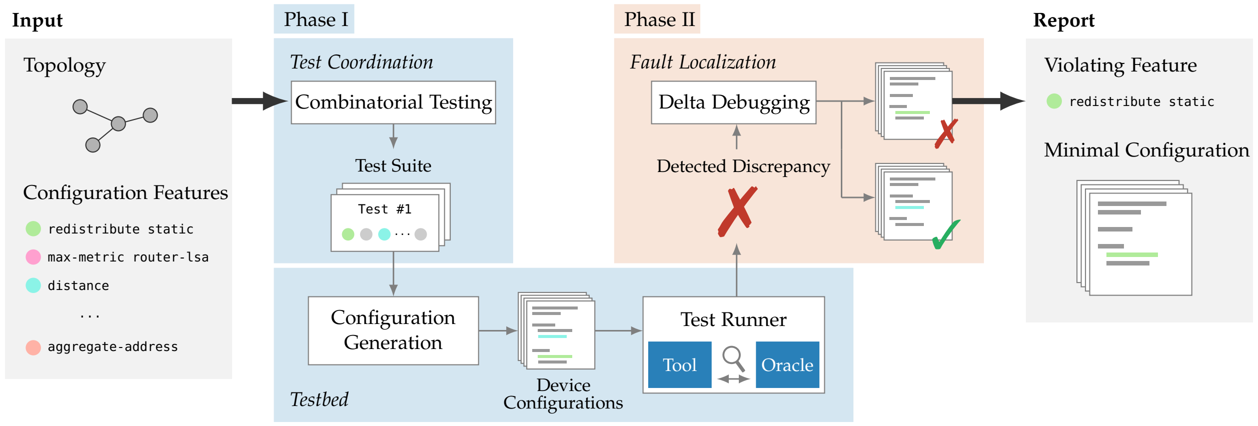


FIGURE 5.3: *Metha* generates a test suite based on the test topology and supplied configuration features. The testbed runs one test after another and compares the computed routing tables of the tool under test to those of an oracle. It then analyzes every discrepancy to localize all the bugs and creates a report for each one of them.

$$\begin{array}{lcl}
\mathit{BGPProcess} & \rightarrow & \texttt{router bgp}\ \mathit{Integer16}\ [\mathit{Options}] \\
\mathit{Options} & \rightarrow & \mathit{Option} \mid \mathit{Options}\ \mathit{Option} \\
\mathit{Option} & \rightarrow & \mathit{Redistribute} \mid \mathit{Neighbor} \mid \mathit{Network} \mid \cdots \\
\mathit{Redistribute} & \rightarrow & \texttt{redistribute}\ \mathit{Source} \\
\mathit{Source} & \rightarrow & \texttt{direct} \mid \texttt{static} \mid \cdots \\
\mathit{Neighbor} & \rightarrow & \texttt{neighbor}\ \mathit{Address}\ \mathit{Property} \\
\mathit{Property} & \rightarrow & \mathit{RemoteAS} \mid \mathit{RouteMap} \mid \cdots \\
\mathit{RemoteAS} & \rightarrow & \texttt{remote-as}\ \mathit{Integer16} \\
\mathit{RouteMap} & \rightarrow & \texttt{route-map}\ \mathit{String}\ \mathit{Direction} \\
\mathit{Direction} & \rightarrow & \texttt{in} \mid \texttt{out}
\end{array}$$

FIGURE 5.4: Partial BNF grammar for device configurations.

5.3 SEARCH SPACE

In this section, we define the search space of all possible configurations. We also show how we reduce the search space by restricting the parameter values used in configuration statements to their *boundary* values.

5.3.1 *Network Configurations*

The search space is given by all possible configurations that one can deploy at the network's routers.

CONFIGURATIONS A device configuration defines the enabled features along with their parameter values. Formally, the set of all possible configurations is defined by a context-free grammar whose terminals consist of feature names and parameter values. To illustrate this, in Figure 5.4 we show a subset of the production rules in Backus-Naur form (BNF). An example configuration derived from this grammar is:

```
router bgp 100
  redistribute static
  neighbor 1.1.1.2 remote-as 50
  neighbor 1.1.1.2 route-map map10 out
```

This configuration defines the AS identifier, the session with a BGP neighbor, the properties of this session, and route redistribution associated with the BGP routing process `100`. Here, `router bgp`, `redistribute`, `neighbor A.B.C.D remote-as`, and `neighbor A.B.C.D route-map` are configuration statements, while the values to their right define their parameter values. We distinguish three types of parameter values:

KEYWORDS are used in configuration statements parameterized by a value drawn from a fixed set of options. For example, the configuration statement `neighbor A.B.C.D route-map` is parameterized by a direction, which is set to either `in` or `out`. For some statements, one can also omit the parameter value altogether, which we model with the designated value $\varnothing$. The statement `redistribute connected`, for example, is parametrized by a value drawn from the set $\{\varnothing$,`subnets`$\}$, and so both the statements `redistribute connected` and `redistribute connected subnets` are valid configuration statements.

INTEGERS are used to define 16- and 32-bit numbers. For example, the configuration statement `router bgp` is parameterized by a 16-bit integer defining the AS number.

STRINGS are used in configuration statements parameterized by custom names. For example, `neighbor A.B.C.D route-map` is parameterized by the route-map's name.

SEMANTIC CONSTRAINTS Besides conforming to the syntax shown in Figure 5.4, configurations must also comply with semantic constraints. For example, consider the following configurations:

```
interface FastEthernet0/0
  ip address 1.1.1.1 255.255.255.0
!
router bgp 100
  neighbor 1.1.1.2 remote-as 50
  neighbor 1.1.1.2 route-map map10 out
!
route-map map10 permit 10
  match ip address prefixList
```

```
interface FastEthernet0/0
  ip address 1.1.1.2 255.255.255.0
!
router bgp 50
  neighbor 1.1.1.1 remote-as 100
```

The top configuration (C_1) defines a BGP process with AS number `100` (Line 4), and declares that announcements sent to its BGP neighbor with IP `1.1.1.2` (Line 5) are processed using route-map `map10` (Line 6). The bottom configuration (C_2) defines a BGP process with AS number 50 (Line 4), and declares `1.1.1.1` in AS `100` as a neighbor (Line 5). These two configurations illustrate two kinds of semantic constraints:

INTRA-DEVICE CONSTRAINTS, which stipulate conditions that must hold on any (individual) configuration. For example, the route-map `map10` used at Line 6 must be defined within the configuration C_1. This constraint holds as `map10` is defined at Line 8.

INTER-DEVICE CONSTRAINTS, which stipulate conditions across multiple configurations. For example, the AS number assigned to neighbor `1.1.1.2` in C_1 at Line 5 must match the AS number declared in C_2 at Line 4. This constraint holds as at both lines the AS number is 50.

Finally, we note that we specify the semantic constraints separately from the syntactic production rules as some are not context-free and thus cannot be encoded in the grammar.

SEARCH SPACE The search space used by *Metha* is defined by the set of configurations that one can deploy at the network's routers. As the set of configurations derived from the grammar is, in general, infinite, we restrict all recursive rules so that its language consists of finitely many configurations. For instance, for the grammar given in Figure 5.4, we fix the set of BGP options (such as route redistribution) that can appear when defining a BGP routing process. Finally, the search space of *Metha* is defined as C^R, where C is the set of all configurations and R is the set of routers. Note that each element of C^R defines a *network-wide configuration*, assigning a configuration from C to each router in R.

5.3.2 *Boundary Values*

The search space is extremely large due to the enormous number of configurations and the exponentially many combinations in which they can be deployed at the routers. To cope with the large set of configurations, we apply a *boundary values* reduction by restricting the parameters to a small set of representative values. The intuition behind this reduction is that most parameter values lead to the same behavior such that testing them individually provides no additional insights.

The reduction to boundary values ensures that various behaviors of a feature are exercised. For example, the Cisco BGP feature `neighbor X.X.X.X maximum-prefix n` terminates the session when the neighbor announces more than `n` prefixes. When randomly choosing `n`, the feature will most likely not come into action. However, with the boundary values, both the minimum and maximum value are tested, ensuring that the feature is at least once active and once not.

For integer parameters, the values are restricted to: the maximum value, the minimum value, and a non-boundary value. For example, for 16-bit integers, which contain all integers in the range $[0, 65535]$, our boundary value reduction selects three values: 0, 65535, and a value x such that $0 < x < 65535$. Similarly, we reduce the values assigned to string parameters by predefining a fixed set of strings.

5.4 EFFECTIVE SEARCH SPACE EXPLORATION

Metha must cover a wide variety of different network configurations to thoroughly test the tool, including many combinations of device features and parameter values. The key challenge is that it is impossible to iterate through every single combination of features and their respective parameter values, even after considering our reduction to boundary values. To address this, *Metha* relies on *combinatorial testing* [83, 84], which is able to uncover all bugs involving a small number of interacting features. In the following, we first provide the relevant background on combinatorial testing, and then we show how *Metha* uses it to effectively test network tools.

5.4.1 *Combinatorial Testing*

Combinatorial testing is a black-box test generation technique which is effective at uncovering *interaction bugs*, i.e., bugs that occur because of multiple interacting features and their parameter values. The main assumption behind combinatorial testing is that interaction bugs are revealed by considering a small number of features and parameter values. In this case, one can generate a test suite, called *combinatorial* test suite, that uncovers all of these bugs.

To use combinatorial testing, one needs to define a specification of the system's parameters and their values:

Definition 5.1 (Combinatorial specification). *A combinatorial specification $\mathcal{S}$ is a tuple (P, V, Δ), where P is a set of parameters, V is a set of values, and $\Delta \colon P \to 2^V$ defines the domain of values $\Delta(x) \subseteq V$ for any parameter $x \in P$.*

For example, the combinatorial specification for a program that accepts three boolean flags as input has parameters $P = \{a, b, c\}$, values $V = \{0, 1\}$, and domains $\Delta(a) = \Delta(b) = \Delta(c) = \{0, 1\}$. A *test case* is a total function $tc \colon P \to V$ mapping parameters to values from their respective domains, i.e., with $P(x) \in \Delta(x)$ for any x. An example test case for our program is $tc = \{a \mapsto 0, b \mapsto 0, c \mapsto 1\}$. In contrast to test cases, a *t-wise combination* maps only some parameters to values:

Definition 5.2 (t-wise combination). *Given a combinatorial specification $\mathcal{S} = (P, V, \Delta)$, a t-wise combination for $\mathcal{S}$ is a function $c \colon Q \to V$ such that $Q \subseteq P$ with $|Q| = t$ and $c(x) \in \Delta(x)$ for any $x \in P$.*

An example pairwise combination (i.e., $t = 2$) for our example is $c = \{a \mapsto 0, b \mapsto 1\}$. We write $\mathcal{C}_t^S$ to denote the set of all t-wise combinations for a given combinatorial specification $\mathcal{S}$. Note that a test case can cover multiple t-wise combinations:

$$comb_t(tc) = \{c \subseteq tc \mid |c| = t\}$$

For instance, our example test case above covers the following three pairwise combinations: $\{a \mapsto 0, b \mapsto 0\}$, $\{a \mapsto 0, c \mapsto 1\}$, and $\{b \mapsto 0, c \mapsto 1\}$.

Definition 5.3 (t-wise combinatorial coverage). *Given a combinatorial specification $\mathcal{S}$, we define the t-wise combinatorial coverage of a test suite T as:*

$$cov_t(T) = \frac{|\bigcup_{tc \in T} comb_t(tc)|}{|\mathcal{C}_t^S|} .$$

A test suite T is called a *t-combinatorial test suite* if $cov_t(T) = 1$. If the assumption that interaction faults are caused by up to t-wise interactions holds, then T finds all bugs. The goal of combinatorial testing is to generate the smallest t-wise combinatorial test suite.

5.4.2 *Combinatorial Testing of Configurations*

In *Metha*, we apply pairwise combinatorial testing to the generation of network configurations. Concretely, we phrase the search space defined in

Section 5.3 as a combinatorial specification $\mathcal{S} = (P, V, \Delta)$ as follows. First, each statement that can appear in the configuration, such as route redistribution or route-map as defined in Section 5.3, defines a configuration feature. We set F to be the set of all configuration features. The set of parameters P is then given by $R \times F$, where R is the set of routers. Namely, the parameters consist of all configuration features one can define in the device configurations.

Second, the domains of values for each configuration feature contain the boundary values that can be used in the given configuration statement, along with the designated value $\perp$, which indicates whether the configuration feature is enabled or not. That is, $\perp$ results in omitting the configuration statement altogether. We note that for configuration statements with multiple parameters, we take the product as the domain of possible values. The Cisco OSPF configuration feature `default-information-originate`, for example, has three optional parameters: `always`, `metric` combined with an integer value, and `metric-type` combined with 1 or 2. After reduction to boundary values this leads to the following three parameters:

$$\begin{aligned}
A &= \{\varnothing, \texttt{always}\} \\
B &= \{\varnothing, \texttt{metric 1}, \texttt{metric 100}, \texttt{metric 1677214}\} \\
C &= \{\varnothing, \texttt{metric-type 1}, \texttt{metric-type 2}\}
\end{aligned}$$

The domain of values for this configuration feature is then given by $\{\perp\} \cup (A \times B \times C)$.

Finally, *Metha* uses the above combinatorial specification to derive a test suite of configurations that covers all pairwise combinations.

5.5 FAULT LOCALIZATION

A discovered discrepancy between the network model and the oracle is only of limited use as the developer still needs to isolate its cause. Often understanding the bug is the most time-consuming part of the debugging process, and fixing it can be done relatively quickly. To help with this, *Metha* pinpoints the configuration features that cause a discrepancy and finds a minimal configuration, i.e., a configuration with as few configuration features enabled as possible. To do this, *Metha* uses *iterative delta debugging*, an extended version of classic delta debugging, which lifts the assumption that a single fault causes failures. This extension is important as network configurations are large and complex, and discrepancies are often caused by multiple faults. In the following, we first introduce classic delta debugging and then present its iterative extension.

5.5.1 *Delta Debugging*

Delta debugging [85] is a well-established fault localization technique, which finds minimal failure-inducing inputs from failing test cases. Below, we present delta debugging in our context, and then define its assumptions and algorithmic steps.

TERMINOLOGY As defined in Section 5.4, a test case tc assigns configuration features F to either parameter values or $\bot$, where $\bot$ indicates that a given feature is disabled (i.e., it is omitted from the configuration). Given a test case tc and features $Q \subseteq F$, we write $tc|_Q$ for the test case obtained by disabling all features in tc that are not contained in Q:

$$tc|_Q(f) = \begin{cases} tc(f) & \text{if } f \in Q \\ \bot & \text{otherwise} \end{cases}$$

Given a failing test case tc, the goal of delta debugging is to find the minimal set Q of features such that $tc|_Q$ fails. We denote the complement of Q by $\bar{Q} = F \setminus Q$.

ASSUMPTIONS Delta debugging relies on three assumptions: *(i)* test cases are *monotone*, i.e., if $tc|_Q$ fails, then for any superset $Q' \supseteq Q$ of features $tc_{Q'}$ also fails; *(ii)* test cases are *unambiguous*, meaning that for a failing test case tc there is a unique minimal set Q that causes the failure; and *(iii)* every subset of features is *consistent*, meaning that for any $Q \subseteq F$, $tc|_Q$ terminates with a definite fail or success result.

ALGORITHM Given a test case tc, delta debugging finds a minimal set of features Q that causes a failure. Initially, Q contains all enabled features in tc, i.e., $Q = \{f \in F \mid tc(f) \neq \bot\}$. Then it applies the following steps:

1. *Split:* Split Q into n partitions $Q_1, \ldots, Q_n$, where n is the current granularity. Test $tc|_{Q_1}, \ldots, tc|_{Q_n}$ for failures. If some $tc|_{Q_i}$ fails, then use Q_i as the new current set of features and continue with step 1.
2. *Complement:* If none of the new tests $tc|_{Q_1}, \ldots, tc|_{Q_n}$ fail, check the complement of each partition by testing $tc|_{\bar{Q}_1}, \ldots, tc|_{\bar{Q}_n}$. If some $tc|_{\bar{Q}_i}$ fails, then use $\bar{Q}_i$ as the new current set of features and continue with step 1.

Algorithm 5.1: Iterative Delta Debugging

```
Input   : Test case $tc$, initially enabled features $Q$ in $tc$.
Output: A set of minimal feature subsets $\mathcal{S} = \{Q_1, \ldots, Q_n\}$.
$\mathcal{S} = \emptyset$
$Queue$ = queue()
put($Queue, Q$)
while $\neg$empty($Queue$) do
    $H$ = head($Queue$)
    if run($tc|_H$) = failure then
        $Q' = minimize(H)$
        for $f$ in $Q'$ do
            put($Queue, H \setminus \{f\}$)
        $\mathcal{S} = \mathcal{S} \cup \{Q'\}$
return $\mathcal{S}$
```

3. *Increase Granularity:* If no smaller set of features is found and $n < |Q|$, then set n to $\min(2n, |Q|)$ and continue with step 1.
4. *Terminate:* If it is not possible to split the current set of features into a smaller set, terminate and return Q.

5.5.2 *Iterative Delta Debugging*

In our setting, test cases are often ambiguous as a discrepancy often arises due to multiple faults in the network model. To this end, we apply the delta debugging algorithm *iteratively* and find all minimal sets of features that cause a given discrepancy. Intuitively, starting with a test case tc with enabled features Q, we first apply the delta debugging steps (given in Section 5.5.1) to find a minimal configuration feature set Q' such that $tc|_{Q'}$ triggers the discrepancy. Then, we generate new test cases $tc_1, \ldots, tc_{|Q'|}$, by disabling a feature from Q in each new test case tc_i, and iteratively apply delta debugging to these. We apply this process repeatedly until no further failing test cases are found. Once *Metha* identifies all minimal sets of configuration features that trigger a given bug, *Metha* creates a minimal configuration for the developer to reproduce it.

We present our iterative delta debugging algorithm in Algorithm 5.1. We start from a set of initially enabled features Q in tc and return all minimal subsets of Q that trigger a discrepancy. We keep all sets of features to be checked in a queue and continue until the queue is empty (Line 2 - Line 4). For every set H of features in the queue, we check if the test case $tc|_H$ triggers a discrepancy (Line 5, Line 6). If this is the case, then we find a minimal subset $Q \subseteq H$ of features that triggers the discrepancy using classic delta debugging, and create new subsets that need to be checked (Line 8, Line 9). For example, if we find a minimal set of features $Q = \{a, b\}$ that triggers the discrepancy, then we check if there are any other minimal sets of features that do not contain a or b (and are thus non-comparable to Q). We note that we generate two new sets of features $H \setminus \{a\}$ and $H \setminus \{b\}$ instead of a single one $H \setminus \{a, b\}$ because there may be overlapping discrepancies. For example, even though we know that b can trigger a discrepancy with a, b might also trigger a discrepancy with another feature c. Finally, the algorithm keeps all found minimal feature subsets and returns them (Line 10, Line 11). We conclude by stating the correctness of our algorithm:

Theorem 5.1. *For any test case tc with enabled features Q, Algorithm 5.1 finds all minimal fault-inducing subsets of features.*

Proof. Assume, for the sake of contradiction, that there is a minimal subset $Q' \subseteq Q$ such that $tc|_{Q'}$ fails which is not returned in $\mathcal{S}$. We check at least one superset of Q' for a failure since we will always check the initial set Q. Assume $C \supseteq Q'$ is a smallest superset of Q' which is checked. By the assumption of monotonicity, $tc|_C$ must fail, therefore we will minimize C. If $C = Q'$, then we must minimize to Q' since Q' is assumed to be minimal, violating the assumption that Q' is not returned by the algorithm. If $Q' \subset C$, then C will either minimize to Q' (again violating the original assumption that Q' is not returned by the algorithm) or to a different minimal subset P. In this case, we generate additional sets to be tested. However, both Q' and P are minimal subsets of C, therefore $Q' \not\subset P$ and $P \not\subset Q'$. Since $Q' \neq P$, we know that there must be an element $e \in P$ which is not in Q', i.e., such that $Q' \subseteq C \setminus \{e\}$. The set $C \setminus \{e\}$ is both strictly smaller than C and will be added to the sets to check by the algorithm in Line 9 and therefore violates our assumption that C was a smallest superset of Q' which is checked. □

RUNTIME The running time of Algorithm 5.1 is $O(|Q|!)$. The worst-case behavior is when the size of the set H of features is reduced by 1 element in each step, introducing $|H - 1|$ new features sets to the set $\mathcal{S}$. To improve the running time, we cache (not shown in Algorithm 5.1) feature sets that have been added to the queue. This strictly reduces the algorithm's running time and yields a worst-case running time complexity of $O(2^{|Q|})$. We note that the running time in practice is reasonable as the reduction of the set H by the delta debugging minimization step (Line 7) is significant (down to $2-3$ elements in practice).

LIMITATIONS As with classic delta debugging, there may be a fault in the interaction between a set of parameters, say a, b, and c, as well as a different fault in the interaction between a subset of these parameters, say a and b. We cannot distinguish these two faults and will only identify the latter fault. However, once the identified fault is fixed, our algorithm will then identify the fault in the interaction among a, b, and c as well, assuming it is still present in the network validation tool.

5.6 SYSTEM

We have fully implemented *Metha* in 7k lines of Python code.[1] This covers the entire testing pipeline from the input, the list of configuration features to be tested and the topology, to the outputs, the bug reports. In the following, we highlight key points of *Metha*'s implementation, which consists of a vendor- and tool-agnostic core that uses runners to interface with the different network analysis and verification tools.

SEMANTIC CONSTRAINTS To run the tests, *Metha* uses a logical topology, which consists of the physical topology extended with logical groupings. These groupings map the routers to BGP ASes and their interfaces to OSPF areas. This trivially ensures that the base configuration meets all the necessary semantic constraints (cf. Section 5.3.1). In a next step, *Metha* starts to randomly assign IP subnets to links and IP addresses to the router interfaces on these links. Specifically, every router is assigned a router ID, which is also assigned to the loopback interface of that router. Finally, *Metha* generates additional resources that are needed to test specific configuration features. For example, *Metha* adds several prefix-lists and static routes which can then be used in the test generation, for example, for a match statement of a route-map and route redistribution, respectively. All these additional resources are generated based on the predefined logical topology. Hence, a prefix-list, for example, will only consist of prefixes that are actually defined in the network, such that a route-map statement using that list for a match will also be reachable.

1 Available at `https://github.com/nsg-ethz/Metha`

TESTING COORDINATION Once *Metha* laid the groundwork, it has to define a test strategy based on the specified configuration features. At the moment, *Metha* supports configuration features pertaining to four categories: static routes, OSPF, BGP and route-maps. As part of that, the system supports additional constructs such as prefix-lists and community-lists. These are currently not tested on their own, but added when needed to test the main features, such as route-maps. *Metha* then uses all features and the logical topology to prepare the parameters to come up with the test suite. To do that, *Metha* passes all the parameters and their possible values to a state-of-the-art combinatorial testing tool: PICT [86]. PICT devises a test suite that consists of a set of tests ensuring complete coverage of all pairwise feature interactions.

CONFIGURATION GENERATION A single test from the PICT test suite is an abstract network configuration. It simply specifies which feature and corresponding value needs to be activated and where (i.e., on which router and, if applicable, at which interface). *Metha* then translates the abstract network configuration to concrete device configurations using a grammar-based approach to ensure lexical and syntactical validity.

Metha implements a large portion of both Cisco IOS and Juniper grammars for which we relied on the respective official command references. This means *Metha* can generate both Cisco IOS and Juniper configurations for the tests. *Metha* even supports to test hybrid networks in which devices of both vendors are used at the same time.

TESTBED *Metha* runs the generated configurations in parallel on both the tool under test and the oracle. After both of them converged, it retrieves the routing tables and compares them. *Metha* is able to test any tool that takes the device configurations as inputs and provides direct access to the computed routing tables out-of-the-box. Otherwise, *Metha* uses tool-specific runners to process the inputs such that they meet the tool's requirements and map the output back to *Metha*'s format. *Metha* comes with runners for three well-known network analysis and verification tools: Batfish [81], NV [87] and C-BGP [32]. For NV, for example, *Metha* first has to compile the simulation program from the network configurations. As a source of ground truth, *Metha* uses a virtualized network running real device images of both Cisco and Juniper routers. It connects to these devices over Telnet and retrieves the routing tables (e.g., `show ip route` for Cisco devices). To ensure full convergence, *Metha* retrieves the routing tables every 10 seconds and proceeds once the tables have not changed for ten consecutive checks. With this setup, *Metha* allows to freely choose any oracle (e.g., hardware testbed) as long as it exposes the computed routing tables.

OUTPUT Finally, *Metha* localizes all bugs within a discovered discrepancy by relying on delta debugging. For every single bug, it generates a report highlighting the observed difference in the routing tables of the tool under test and the oracle, such as a mismatch in a route's metric or a missing route. This helps the developer understand the expected behavior. In addition, it identifies the configuration statements required to trigger the bug and comes up with a minimal network configuration to reproduce the bug. This allows *Metha* to provide actionable feedback to the developers of the tool, helping them to faster locate and understand the bug. The minimal configuration example can also be used as an extra test case for traditional system testing.

5.7 EVALUATION

In this section, we evaluate *Metha* to address the following research questions:

RQ1 How does *Metha*'s semantical configuration generation, the search space reduction using boundary values and the test suite from combinatorial testing contribute to *Metha*'s effectiveness? We show that *Metha* finds 20 bugs and achieves a higher combinatorial coverage than the random baseline, which only discovers 3 bugs with the same number of tests (Section 5.7.1).

RQ2 How many test cases does *Metha* need to localize all bugs in a single discrepancy between the tool under test and the oracle? *Metha* requires on average 14.1 test cases to isolate all the bugs causing a discrepancy (Section 5.7.2).

RQ3 Is *Metha* practical? We ran *Metha* on three different state-of-the-art network analysis and verification tools and found a total of 62 bugs, 59 of them have been confirmed by the respective developers (Section 5.7.3).

5.7.1 *Comparison to Random Baseline*

We begin our evaluation by studying how the three components of *Metha* contribute to its effectiveness. To this end, we compare a random baseline to three versions of *Metha*: step-by-step, we enable each component starting with semantic *Metha*, then we add the reduction to boundary values, and finally, we use full *Metha* using combinatorial testing to define a test suite. The results show that the semantical configuration generation is the most fundamental part of *Metha*. Reducing the parameters to boundary values and applying combinatorial testing help to find additional bugs as both manage to increase the combinatorial coverage.

In the following, we introduce the four approaches:

RANDOM BASELINE The random baseline relies on random syntactic test generation, which means that it uses a traditional grammar-based fuzzing approach. Thanks to the grammar, the configurations generated by the baseline are lexically- and syntactically-valid, but they are not necessarily semantically-valid: the baseline generates device configurations that are parseable and look realistic. However, the configurations might not always be practical: for example, referenced route-maps and prefix-lists do not always exist, and IP addresses on interfaces might not match those of their neighbors. Inter- and intra-device dependencies are not factored in.

SEMANTIC METHA The initial *Metha* approach implements random semantic test generation. Similar to the random baseline, it uses a grammar-based fuzzing approach with the only difference that it ensures semantical validity within the configuration: while, for example, interface costs are completely random, other values are more constrained based on inter- and intra-device dependencies. This approach ensures, for example, that only defined route-maps are referenced, and that BGP sessions are configured with matching parameters.

BOUNDED METHA The bounded approach adds the reduction to boundary values as introduced in Section 5.3.2 to semantic *Metha*. This means instead of assigning completely random numeric values, the approach reduces the allowed values to three options: the minimum, the maximum, and a "normal" value, randomly chosen between the two extremes.

FULL METHA Finally, we run the full testing system. We add combinatorial testing as introduced in Section 5.4 to define a test suite that maximizes combinatorial coverage on top of the semantic configuration generation and the boundary values reduction.

Approach	# Discovered Bugs
Random Baseline	3
Semantic *Metha*	16
Bounded *Metha*	17
Full *Metha*	**20**

TABLE 5.2: Every component of *Metha* allows it to find more bugs with the same number of test runs.

EXPERIMENT SETUP We ran all four approaches for the same number of tests and used them to test Batfish [81]. Whenever one of them detected a discrepancy between Batfish and the oracle, we applied the full fault localization procedure as described in Section 5.5 to detect the underlying bugs and the features causing it. Thanks to that, we are able to detect duplicates and count only the unique bugs that each approach discovered.

For all the tests, we used the same simple topology consisting of four routers connected in a star topology and tested configuration features belonging to the following four categories: static routes, BGP, OSPF, and route-maps. For the entire experiment, we used Cisco IOS configurations. For the given configuration features, combinatorial testing generated a test suite consisting of 1 794 tests. While the full *Metha* approach followed the test suite, the other approaches randomly chose the active configuration statements for every single test.

RESULTS Table 5.2 shows the number of unique bugs that every approach found within the 1 794 test runs. The full *Metha* approach detected 20 unique bugs, while the random baseline only found 3 bugs. The semantic configuration generation is the most fundamental component of *Metha*. It comes as no surprise as without semantical validity, many of the configurations do not allow for any meaningful control-plane computations and will not fully exercise the network model of the tool under test.

Boundary values and combinatorial testing allow finding 1 and 3 additional bugs within the 1 794 test runs, respectively. This is because both approaches achieve higher combinatorial coverage and therefore test a wider variety of features. These results show that the boundary values reduction strikes a good balance between testing different parameter values, while keeping the search space tractable. It is important to note that the detected bugs are inclusive, meaning that full *Metha* detected all 17 bugs

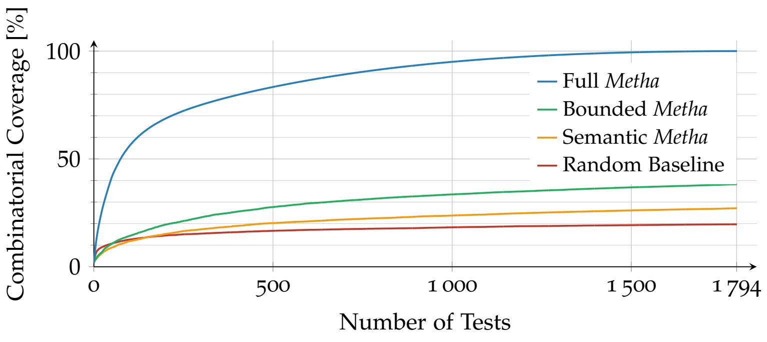


FIGURE 5.5: The achieved combinatorial coverage increases with every single component of *Metha*. Full *Metha* achieves complete combinatorial coverage.

that bounded *Metha* detected and 3 additional bugs. There is one exception: the baseline found a bug in the parser, which the other approaches did not find.

The random baseline is strong at discovering parser bugs since that is where grammar-based fuzzing excels. Two out of its three discovered bugs are parser bugs. In both cases, the problem was an, according to the specification, unsigned 32-bit integer being parsed as a signed integer. For example, `ip ospf 100 area 3933914791` could not be parsed. *Metha* did not catch this bug as it uses fixed area numbers as part of the logical topology. By adding the area numbers to the set of configuration features being tested, *Metha* also finds this bug.

Figure 5.5 shows the combinatorial coverage achieved by the four approaches, i.e., it shows the pairwise feature combinations covered during testing. We focus on feature instead of code coverage for two reasons: First, one can easily achieve high code coverage with random, semantically-invalid configurations. Second, code coverage is specific to the tool under test and makes it difficult to compare. To measure the combinatorial coverage of the random baseline and semantic *Metha*, we partitioned the input space in the same manner as we did for bounded *Metha*, i.e., into minimum, maximum, and middle values. Any configuration which did not specifically use the minimum or maximum value for a parameter was then considered as a middle configuration. *Metha* achieves full combinatorial coverage by design as it is guaranteed by combinatorial testing. These results underline the importance of semantically-valid configurations. While both the random baseline, which relies on syntactically-valid configurations, and semantic *Metha* achieve a similar combinatorial coverage, semantic *Metha* finds many more bugs as its configuration actually ensures control-plane computations.

performance Running a single test case took an average of two minutes. We run both the tool under test as well as the virtualized testbed in parallel and found that most of the time is spent waiting on the testbed to converge. The generation of a combinatorial test suite with PICT for the baseline network with 4 routers took an average of 6 minutes. Over the entire test suite, this time is negligible. Running the entire setup took us several days. The runtime depends highly on the number of discrepancies and the number of bugs causing them.

5.7.2 *Fault Localization*

Whenever *Metha* detects a discrepancy between the routing tables of the tool under test and those of the oracle, it goes into fault localization to isolate all independent bugs. Fault localization relies on delta debugging (cf. Section 5.5) which creates additional test cases to identify the configuration statements causing the bugs. In the following, we evaluate its overhead, i.e., the number of additional test cases *Metha* had to create.

experiment setup For this experiment, we ran *Metha* using the same topology as before and tested the full set of configuration features. Whenever *Metha* detected a discrepancy, we recorded the number of additional test cases required to find all independent bugs and the number of discovered bugs.

results On average, *Metha* used 14.1 additional test cases to locate all bugs within a test case. The number of additional test cases ranged from as low as 7, to localize a single bug, up to as high as 58, to localize 5 independent bugs. The number of additional test cases mostly depends on the number of independent bugs within a single detected discrepancy. The number of configuration statements that actually cause the bug plays a minor role. Also, we have observed that the detected bugs are all caused by a few configuration statements (one or two), even though multiple configuration statements were active during the tests. This confirms the observation that bugs are often caused by the interaction of few features [83, 88] and shows that combinatorial testing is a useful technique in this setting.

	Bugs		Type	
	discovered	confirmed	crash	silent
Batfish [81]	29	29	5	24
NV [87]	30	30	5	25
C-BGP [32]	3	?	0	3

TABLE 5.3: Bugs discovered by *Metha* for Batfish, NV and C-BGP and their type.

5.7.3 *Real Bugs*

In addition, we showcase our end-to-end implementation of *Metha* by testing three different network analysis and verification tools: Batfish [81], NV [87], and C-BGP [32]. We show that *Metha* finds real bugs and report them in Table 5.3.

EXPERIMENT SETUP We ran *Metha* for several days on all three tools and with several different setups. Batfish is the most complete and advanced tool as it can handle configurations of many different vendors and supports a wide variety of configuration features. NV itself is an intermediate language for control-plane verification that allows to build models of any routing protocols and their configurations. It provides simulation and verification abilities. We tested the simulation only, the discovered bugs, however, most likely also exist in the verification part as both rely on the same network model. For Batfish and NV, we used both Cisco IOS and Juniper configurations. C-BGP has its own configuration language.

RESULTS As shown in Table 5.3, *Metha* found a total of 62 bugs. The developers of both Batfish and NV confirmed the discovered bugs to be real bugs. To better understand the nature of the bugs, we classified them by their type (i.e., whether they lead to a crash or go unnoticed) and by the configuration feature category itself (e.g., OSPF). Only a few of the bugs produce a clear error. This is most likely also because these are noticed more often and reported. The large majority of the bugs are silent semantic bugs which are extremely difficult to notice. These are the sneakiest bugs and can lead to false analyses and answers by the verifier.

As shown in Table 5.4, the bugs include all the configuration features discussed in Section 5.1 and affect the analysis of commonly used features,

	Feature Category			
	OSPF	BGP	route-filter	other
Batfish [81]	10	10	9	0
NV [87]	13	9	7	1
C-BGP [32]	1	1	1	0

TABLE 5.4: Classification of the discovered bugs by the affected feature category.

such as route redistribution and aggregation, and named communities. The bugs are distributed quite evenly among all tested parts of the network model. We did not find one specific protocol or configuration feature that is especially error-prone.

5.8 DISCUSSION

In this section, we discuss topology requirements and the benefits of a virtualized testbed, and we explain the scope of the *Metha*'s test suite.

WHAT ABOUT THE TESTBED? *Metha* detects bugs by looking for discrepancies between the tool under test and an oracle. For the oracle, *Metha* uses a testbed running real router firmwares. The testbed just needs to be large enough to fully exercise all configuration features. Normally, a small testbed of few routers suffices and also helps speed up the testing. In this chapter, we rely on a virtualized testbed. To use a physical testbed instead, one simply has to change the SSH/Telnet configurations to connect to the physical devices.

A virtualized testbed comes with several advantages. It provides more flexibility in terms of the settings one can test and the time needed to setup. For example, there is no re-wiring needed to test different topologies. In addition, it is very simple to test the same topology with a different device category or with devices from another vendor: one simply has to exchange the router image.

WHAT ABOUT MORE TARGETED TESTS? *Metha*'s test suite can be adjusted to the users' requirements by restricting the set of configuration features, adjusting the number of values per feature, and changing the number of interacting features. The tests required to cover the search space mainly depend on the number of values per feature and the number of simultaneous feature interactions, while the set of features is secondary. By default *Metha* tests three values per feature and considers pairwise interactions. This choice strikes a good balance between the number of tests required and thorough testing, as our results confirm: *Metha* found all bugs that the random approaches discovered with fewer tests, despite using "only" the boundary values; and all discovered bugs are caused by one or two interacting configuration features, despite considering interactions of more than just two features.

Metha does not replace traditional unit and system testing, but provides an additional way to find latent bugs anywhere in the system. The advantage of *Metha* is that it requires minimal developer involvement and can be run alongside traditional tests without any additional effort. If desired one can run extensive tests by considering more elaborate feature interactions and more than three values per feature. Often with fuzz testing, one just lets the testing system run indefinitely and collect bug reports.

5.9 RELATED WORK

In this section, we first discuss current network analysis and verification tools. Then, we survey related work on testing static analyzers and verifiers, the various testing initiatives in the field of networking, delta debugging, and fuzz testing.

NETWORK ANALYZERS & VERIFIERS Our work aims to help network operators better understand the capabilities of network analysis and verification tools and facilitates the development of such tools through thorough testing. Over the years, we have seen a rise in tools that simulate networks [32], analyze networks [26, 27, 30, 36], and verify properties of networks and their configurations [16, 34, 40, 41]. All of these tools have in common that they rely on a network model. Any bug or inaccuracy that exists within that network model undermines the soundness of the tools' results and analyses.

In contrast, CrystalNet [35] is a cloud-scale, high-fidelity network emulator running real network device firmwares instead of relying on a network model. Hence, it accurately resembles the real network (e.g., vendor-specific behaviors and bugs in device firmwares are captured).

TESTING ANALYZERS AND VERIFIERS The problem of ensuring the correctness of analysis and verification tools is not specific to networks. In the field of static analysis, several works exist that pursue the same goal. Bugariu et al. [89] apply a unit testing approach, meaning they do not test the entire system but components thereof which simplifies the test generation. Since *Metha* treats the tool under test as a black box, it cannot test certain components separately. Cuoq et al. [90] randomly generate input programs. This technique is mostly effective at testing the robustness of the analyzers. Similar to *Metha*, Andreasen et al. [91] apply delta debugging to find small input programs that help developers understand the bug faster.

TESTING IN NETWORKING Prior work on testing in networking has mainly focused on testing the network and its forwarding state [92] and SDN controllers [93, 94, 95].

Closest to *Metha* is Hoyan [96], a large-scale configuration verifier, in which the results of the verifier (i.e., network model) are continuously compared to the actual network for inaccuracies. It does so during operation and only covers cases that have actually occurred in the network. *Metha*, in contrast, proactively detects the bugs before deployment. This helps operators gain trust in their tools as they can assess the tools' capabilities before using them in production.

DELTA DEBUGGING In automated testing tools, delta debugging is a well-established technique [85, 97] that allows to automatically reduce a failing test case to the relevant circumstances (e.g., lines of code or input parameters). Over the years, researchers came up with several extensions to the general delta debugging algorithm, such as a hierarchical approach [98] that takes the structure of the inputs into account. It first explores the more important inputs allowing to prune larger parts of the input space and hence, requiring fewer test cases.

Traditional delta debugging finds one bug at a time even if the test case is ambiguous and exhibits multiple independent bugs. The developer then fixes one bug and reruns delta debugging to find the next. *Metha* automatically detects the causes of all independent bugs without developer involvement.

FUZZ TESTING Fuzz testing [99, 100] is an umbrella term for various testing techniques relying on "randomized" input generation. *Metha* uses a form of grammar-based fuzzing. Due to the complex dependencies within network-wide configurations, *Metha* first builds a basic configuration structure to ensure semantical validity. Then, it uses fuzzing to test different feature combinations restricted to that structure.

5.10 CONCLUSION

In this chapter, we presented *Metha*, a system that helps network operators understand when they can rely on the analyses of their network validation tools and when they cannot. At the same time, *Metha* helps the developers of these tools build more accurate network models by providing them with actionable reports about all discovered inaccuracies, including minimal configuration examples to reproduce them. *Metha* does so by generating a wide variety of network configurations according to a test suite defined through combinatorial testing. We implemented *Metha* and evaluated it on three state-of-the-art tools. In all tools, *Metha* discovered a total of 62 bugs, 59 of them have been confirmed by the developers.

6 CONCLUSION AND OUTLOOK

In this dissertation, we developed three systems to assist network operators in understanding and reasoning about their networks. In particular, we focused on three aspects of network understanding: *(i)* understanding a network's momentary forwarding behavior; *(ii)* understanding a network's configuration and the specification it enforces; and *(iii)* understanding the capabilities and the accuracy of network validation tools.

In Chapter 3, we introduced *Net2Text*, a system that helps network operators understand their network's behavior by explaining the network-wide forwarding state. Based on the operators' natural language queries, *Net2Text* extracts the relevant information from the forwarding state and the traffic statistics, and produces succinct summaries. We demonstrated that even for large networks with hundreds of routers and full routing tables, *Net2Text* generates high-quality summaries in a few seconds only.

Net2Text allows network operators to focus on running a safe and reliable network by freeing them from the time-consuming tasks of gathering low-level network data and extracting the relevant high-level insights.

In Chapter 4, we presented *Config2Spec*, a system that assists network operators in understanding their network's configuration by automatically mining the specification it enforces. That specification is made up of *all* policies that hold under the failure model provided by the operator (e.g., up to k failures) and *only* those. *Config2Spec*'s approach relies on a novel combination of data-plane analysis and control-plane verification. While data-plane analysis allows to prune the set of candidate policies quickly, control-plane verification is able to validate the remaining candidates efficiently. We showed in an extensive evaluation that *Config2Spec* scales to realistic networks with more than one hundred routers.

Config2Spec allows network operators to understand better the policies their configuration enforces. Using the mined specification, operators can detect misconfigurations by identifying discrepancies between the specification and their intention. In addition, the mined specification can serve as input for configuration verification and synthesis tools.

In Chapter 5, we introduced *Metha*, a system that automatically tests network validation tools to assess their capabilities and identifies subtle bugs and inaccuracies in their underlying network models. *Metha* relies on black-box differential testing and uses a variant of delta-debugging to identify the configuration features that cause the inaccuracies. We demonstrated *Metha*'s effectiveness by finding 62 bugs across three popular network validation tools, of which 59 have been confirmed by the tools' developers.

Metha helps network operators understand the strengths and weaknesses of their network validation tools. This allows operators to use their tools with more confidence. In addition, *Metha* is also useful for the developers as they can directly integrate *Metha* in their development life cycle to automatically identify implementation bugs.

6.1 OPEN PROBLEMS

We see open problems and opportunities for future work in a number of areas of network understanding.

6.1.1 *Dealing with Noisy Data*

We built *Net2Text* under the assumption that we have access to the correct and complete forwarding state of a network. However, often this is not possible, or it comes at an unjustifiable cost. We see an opportunity in relaxing this assumption and devising methods that are able to deal with incomplete and inconsistent data. One way to approach this problem is to use regularization. Intuitively speaking, instead of trying to incorporate all, potentially inconsistent data points in the summary, one omits those that do not fit nicely with the rest. Such an approach can also help detect problems in the network by identifying anomalous data points and automatically calling the operator's attention to them.

6.1.2 *Devising New "Human-Network" Interfaces*

While the natural language interface of *Net2Text* found favor with network operators and showed potential to simplify their daily work, we should not stop there. Natural language is just one of many ways to interact with the network. For example, one could also think of graphical user interfaces in which operators select the devices they are interested in or in which operators are able to draw the flows they want to allow in the network. At the same time, one also has to consider the outputs of tools like *Net2Text* or *Config2Spec* and assess in which situations natural language summaries or rather a graph are favorable.

6.1.3 *Incorporating Additional Data*

Currently, our two systems *Net2Text* and *Config2Spec*, make use of a network's forwarding state, traffic statistics, and configurations. We see opportunities for including additional data sources such as routing tables and control-plane logs. For example, the control-plane logs could serve as input to a provenance tool that helps understand how the current network state came about. In addition, we see opportunities to combine and correlate the insights across multiple data sources to, for example, explain that a shift in the traffic was caused by a route change.

6.1.4 *Supporting Richer Specifications*

Config2Spec mines data-plane specifications, which means that the policies which it learns from the configurations all concern the way traffic is forwarded. While reachability, isolation, waypointing, and load balancing are certainly of concern for network operators, they are not the only relevant policies when managing a network. Another important class of policies are control-plane policies, which govern how the routes in the network are computed and disseminated. For example, operators might enforce transit policies to specify which routes can be exported to a specific neighbor. So far, control-plane policies have been neglected by the community and we see here the opportunity for further work by applying similar techniques as in *Config2Spec*.

6.1.5 *Detecting Configuration Bugs*

Config2Spec mines the network's specification as it is enforced by the network's configuration. This does not necessarily represent the specification intended by the operators, as they could have made a mistake. Misconfigurations are indeed quite common due to the difficult and manual configuration process, as we have mentioned in Chapter 2. In theory, network operators could go through the mined specification policy-by-policy and see whether it conforms to their intention. However, as network specifications can easily consist of thousands of policies, manually checking is not practical. Hence, we see opportunities to build tools that automatically detect potential bugs and present them to the operators. Hence, similar to Self-Starter [101], one could rely on the *bugs as outliers* paradigm [102].

BIBLIOGRAPHY

[1] BBC. *United Airlines jets grounded by computer router glitch*. https://www.bbc.com/news/technology-33449693. 2015.

[2] Financial Times. *Visa's European payment systems back up after outage*. https://www.ft.com/content/d95698a2-65b3-11e8-90c2-9563a0613e56. 2018.

[3] Tagesanzeiger. *"Haben Sie Bargeld?" Schweizer konnten nicht per Karte zahlen*. https://www.tagesanzeiger.ch/wirtschaft/standardschweizweite-stoerung-zahlterminals-funktionieren-nicht/story/16758089. 2019.

[4] SWI swissinfo.ch. *Swisscom boss says sorry for network failure*. https://www.swissinfo.ch/eng/swisscom-boss-says-sorry-for-network-failure/46784822. 2021.

[5] Business Insider. *Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales*. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7. 2018.

[6] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. "Automatic Life Cycle Management of Network Configurations". In: *ACM SelfDN*. Budapest, Hungary, 2018.

[7] Ryan Beckett and Ratul Mahajan. "Putting network verification to good use". In: *ACM Hotnets*. Princeton, NJ, USA, 2019.

[8] Tony Bates, Philip Smith, and Geoff Huston. *CIDR REPORT for 27 Jul 21*. https://www.cidr-report.org/as2.0/. 2008.

[9] Edsger W Dijkstra et al. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), 269.

[10] John Moy. *OSPF Version 2*. STD 54. http://www.rfc-editor.org/rfc/rfc2328.txt. RFC Editor, 1998.

[11] David Oran. *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142. http://www.rfc-editor.org/rfc/rfc1142.txt. RFC Editor, 1990.

[12] Charles Hedrick. *Routing Information Protocol*. RFC 1058. `http://www.rfc-editor.org/rfc/rfc1058.txt`. RFC Editor, 1988.

[13] Yakov Rekhter, Tony Li, and Susan Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. `http://www.rfc-editor.org/rfc/rfc4271.txt`. RFC Editor, 2006.

[14] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "Network-wide Configuration Synthesis". In: *CAV*. Heidelberg, Germany, 2017.

[15] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. "Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations". In: *ACM SIGCOMM*. Florianópolis, Brasil, 2016.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *ACM SIGCOMM*. Los Angeles, CA, USA, 2017.

[17] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. "Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks". In: *ACM POPL*. Paris, France, 2017.

[18] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. "Composing Software-Defined Networks". In: *USENIX NSDI*. Lombard, IL, USA, 2013.

[19] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. "Snowcap: Synthesizing Network-Wide Configuration Updates". In: *ACM SIGCOMM*. Virtual Event, NY, USA, 2021.

[20] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. "Merlin: A Language for Provisioning Network Resources". In: *ACM CoNEXT*. Sydney, Australia, 2014.

[21] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. "PGA: Using Graphs to Express and Automatically Reconcile Network Policies". In: *ACM SIGCOMM*. London, United Kingdom, 2015.

[22] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. "Frenetic: A Network Programming Language". In: *ACM ICFP*. Tokyo, Japan, 2011.

[23] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion". In: *USENIX NSDI*. Renton, WA, USA, 2018.

[24] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. "Synthesis of Fault-Tolerant Distributed Router Configurations". In: *ACM SIGMETRICS*. Irvine, CA, USA, 2018.

[25] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. "AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations". In: *ACM CoNEXT*. Barcelona, Spain, 2020.

[26] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. "A General Approach to Network Configuration Analysis." In: *USENIX NSDI*. Oakland, CA, USA, 2015.

[27] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. "Fast Control Plane Analysis using an Abstract Representation". In: *ACM SIGCOMM*. Florianópolis, Brasil, 2016.

[28] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. "Tiramisu: Fast Multilayer Network Verification". In: *USENIX NSDI*. Santa Clara, CA, USA, 2020.

[29] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks." In: *USENIX NSDI*. Seattle, WA, USA, 2014.

[30] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. "Checking Beliefs in Dynamic Networks." In: *USENIX NSDI*. Oakland, CA, USA, 2015.

[31] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. "Plankton: Scalable Network Configuration Verification through Model Checking". In: *USENIX NSDI*. Santa Clara, CA, USA, 2020.

[32] Bruno Quoitin and Steve Uhlig. "Modeling the Routing of an Autonomous System with C-BGP". In: *IEEE Network* 19.6 (2005).

[33] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. "Efficient Network Reachability Analysis Using a Succinct Control Plane Representation". In: *USENIX OSDI*. Savannah, GA, USA, 2016.

[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Brighten Godfrey. "Veriflow: Verifying Network-Wide Invariants in Real Time". In: *USENIX NSDI*. Lombard, IL, USA, 2013.

[35] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. "CrystalNet: Faithfully Emulating Large Production Networks". In: *ACM SOSP*. 2017.

[36] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *USENIX NSDI*. San Jose, CA, USA, 2012.

[37] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. "Real Time Network Policy Checking using Header Space Analysis". In: *USENIX NSDI*. Lombard, IL, USA, 2013.

[38] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A Seshia, and George Varghese. "ddNF: An Efficient Data Structure for Header Spaces". In: *HVC*. Haifa, Israel, 2016.

[39] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. "Debugging the Data Plane with Anteater". In: *ACM SIGCOMM*. Toronto, ON, Canada, 2011.

[40] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. "Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver". In: *ACM OOPSLA*. Amsterdam, Netherlands, 2016.

[41] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. "NV: An Intermediate Language for Verification of Network Control Planes". In: *ACM PLDI*. London, UK, 2020.

[42] Benoit Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. `http://www.rfc-editor.org/rfc/rfc3954.txt`. RFC Editor, 2004.

[43] Peter Phaal, Sonia Panchen, and Neil McKee. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176. http://www.rfc-editor.org/rfc/rfc3176.txt. RFC Editor, 2001.

[44] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. "BGP Routing Stability of Popular Destinations". In: *ACM IMC*. Marseille, France, 2002.

[45] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[46] Henry Wadsworth Gould. *Combinatorial Identities: A Standardized Set of Tables Listing 500 Binomial Coefficient Summations*. Morgantown, 1972.

[47] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. "The Internet Topology Zoo". In: *IEEE JSAC* 29.9 (2011), 1765.

[48] CAIDA. *AS Organizations Dataset, 2017-04-01*. http://www.caida.org/data/as-organizations.

[49] CAIDA. *BGPStream*. https://bgpstream.caida.org/.

[50] Jaeyoung Choi, Jong Han Park, Pei-chun Cheng, Dorian Kim, and Lixia Zhang. "Understanding BGP Next-Hop Diversity". In: *IEEE Global Internet Symposium*. Shanghai, China, 2011.

[51] Quagga. *Quagga Software Routing Suite*. https://quagga.net/. 2021.

[52] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "FlowRadar: A Better NetFlow for Data Centers." In: *USENIX NSDI*. Santa Clara, CA, USA, 2016.

[53] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. "Stroboscope: Declarative Network Monitoring on a Budget". In: *USENIX NSDI*. Renton, WA, USA, 2018.

[54] Olivier Tilmans, Tobias Bühler, Stefano Vissicchio, and Laurent Vanbever. "Mille-Feuille: Putting ISP Traffic under the scalpel". In: *ACM Hotnets*. Atlanta, GA, USA, 2016.

[55] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. "The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance". In: *ACM SIGCOMM*. Florianópolis, Brasil, 2016.

[56] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. "Diagnosing Missing Events in Distributed Systems with Negative Provenance". In: *ACM SIGCOMMM*. Chicago, IL, USA, 2014.

[57] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. "Distributed Time-aware Provenance". In: *VLDB*. Riva del Garda, Trento, Italy, 2013.

[58] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. "Efficient Querying and Maintenance of Network Provenance at Internet-Scale". In: *ACM SIGMOD*. Indianapolis, IN, USA, 2010.

[59] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. "OFRewind: Enabling Record and Replay Troubleshooting for Networks". In: *USENIX ATC*. Portland, OR, USA, 2011.

[60] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences". In: *ACM SIGCOMMM*. Chicago, IL, USA, 2014.

[61] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. "Minimizing Faulty Executions of Distributed Systems". In: *USENIX NSDI*. Santa Clara, CA, USA, 2016.

[62] Azzam Alsudais and Eric Keller. "Hey Network, Can You Understand Me?" In: *IEEE INFOCOM Workshop on Software-Driven Flexible and Agile Networking*. Atlanta, GA, USA, 2017.

[63] Flavio Esposito, Jiayi Wang, Chiara Contoli, Gianluca Davoli, Walter Cerroni, and Franco Callegati. "A Behavior-Driven Approach to Intent Specification for Software-Defined Infrastructure Management". In: *IEEE NFV-SDN*. Verona, Italy, 2018.

[64] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, and Sanjay G Rao. "Hey, Lumi! Using Natural Language for Intent-Based Network Management". In: *USENIX ATC*. Virtual Event, USA, 2021.

[65] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. "Refining Network Intents for Self-Driving Networks". In: *ACM SelfDN*. Budapest, Hungary, 2018.

[66] Ali Kheradmand. "Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns". In: *ACM SOSR*. San Jose, CA, USA, 2020.

[67] Min Cheng. *Small*. `https://github.com/jayvischeng/Small/tree/master/ServerData2`. 2015.

[68] Axel van Lamsweerde. "Formal Specification: a Roadmap". In: *ACM FOSE*. Limerick, Ireland, 2000.

[69] Claire Le Goues and Westley Weimer. "Specification Mining With Few False Positives". In: *TACAS*. York, United Kingdom, 2009.

[70] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R Larus. "Debugging Temporal Specifications with Concept Analysis". In: *ACM PLDI*. San Diego, CA, USA, 2003.

[71] Glenn Ammons, Rastislav Bodík, and James R Larus. "Mining Specifications". In: *ACM POPL*. Portland, OR, USA, 2002.

[72] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Elsevier Science of Computer Programming* 69.1-3 (2007), 35.

[73] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. "On Static Reachability Analysis of IP Networks". In: *IEEE INFOCOM*. Miami, FL, USA, 2005.

[74] Theophilus Benson, Aditya Akella, and David A. Maltz. "Mining Policies from Enterprise Network Configuration". In: *ACM IMC*. Chicago, IL, USA, 2009.

[75] Theophilus Benson, Aditya Akella, and David A Maltz. "Unraveling the Complexity of Network Management". In: *USENIX NSDI*. Boston, MA, USA, 2009.

[76] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. "Management Plane Analytics". In: *ACM IMC*. Tokyo, Japan, 2015.

[77] Yanjun Wang, Chuan Jiang, Xiaokang Qiu, and Sanjay G Rao. “Learning Network Design Objectives Using a Program Synthesis Approach”. In: *ACM Hotnets*. Princeton, NJ, USA, 2019.

[78] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. “NetKAT: Semantic Foundations for Networks”. In: *ACM POPL*. San Diego, CA, USA, 2014.

[79] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. “Combinatorial Sketching for Finite Programs”. In: *ACM ASPLOS*. San Jose, CA, USA, 2006.

[80] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. “Sketching Concurrent Data Structures”. In: *ACM PLDI*. Tucson, AZ, USA, 2008.

[81] Intentionet. *Batfish*. `https://github.com/batfish/batfish`. Commit: 95099bc5ad77af57d92c484e2e5634827f63e724. 2020.

[82] Inc. Cisco Systems. *Redistributing Routing Protocols*. `https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html`. Accessed: 2020-09-12. 2012.

[83] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Transactions on Software Engineering* 30.6 (2004).

[84] Linghuan Hu, W Eric Wong, D Richard Kuhn, and Raghu N Kacker. “How does combinatorial testing perform in the real world: an empirical study”. In: *Empirical Software Engineering* 25.4 (2020).

[85] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002).

[86] Microsoft. *PICT - Pairwise Independent Combinatorial Testing*. `https://github.com/microsoft/pict`. 2020.

[87] Nick Giannarakis. *NV - An Intermediate Language for Network Verification*. `https://github.com/NetworkVerification/nv`. Commit: d058c4ce5c1549ad4e22d97cb01b8ea19d07741c. 2020.

[88] W. Eric Wong, Xuelin Li, and Philip A. Laplante. “Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures”. In: *Journal of Systems and Software* 133 (2017).

[89] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. "Automatically Testing Implementations of Numerical Abstract Domains". In: *ACM ASE*. Montpellier, France, 2018.

[90] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. "Testing Static Analyzers with Randomly Generated Programs". In: *NFM*. Norfolk, VA, USA, 2012.

[91] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. "Systematic Approaches for Increasing Soundness and Precision of Static Analyzers". In: *ACM SOAP*. Barcelona, Spain, 2017.

[92] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. "Automatic Test Packet Generation". In: *ACM CoNEXT*. Nice, France, 2012.

[93] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. "A NICE Way to Test OpenFlow Applications". In: *USENIX NSDI*. San Jose, CA, USA, 2012.

[94] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks". In: *ACM PLDI*. Edinburgh, United Kingdom, 2014.

[95] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. "Correct by Construction Networks Using Stepwise Refinement." In: *USENIX NSDI*. Boston, MA, USA, 2017.

[96] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tianx, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. "Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN". In: *ACM SIGCOMM*. Virtual Event, NY, USA, 2020.

[97] Andreas Zeller. "Yesterday, my program worked. Today, it does not. Why?" In: *ACM SIGSOFT FSE-7* (1999).

[98] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging". In: *ICSE*. Shanghai, China, 2006.

[99] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. "Fuzzing: Breaking Things with Random Inputs". In: *The Fuzzing Book*. Accessed: 2020-09-12. Saarland University, 2020.

[100] Patrice Godefroid. "Fuzzing: Hack, Art, and Science". In: *Communications of the ACM* 63.2 (2020).

[101] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. "Finding Network Misconfigurations by Automatic Template Inference". In: *USENIX NSDI*. Santa Clara, CA, USA, 2020.

[102] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code". In: (2001).