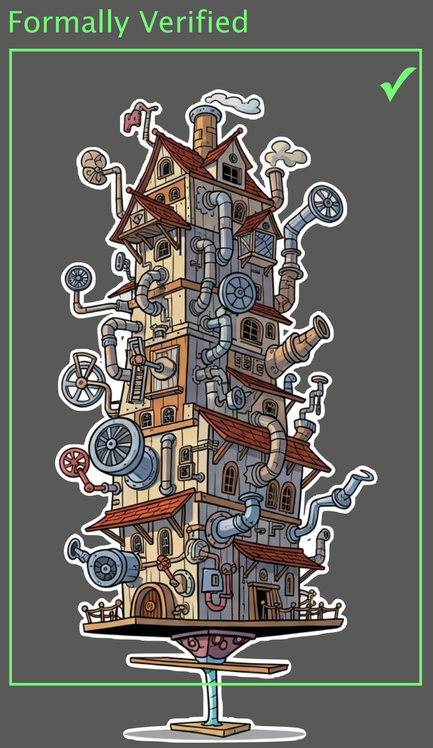
Verifying configurations was the easy part

My ongoing quest toward correct network operations



Laurent Vanbever

nsg.ethz.ch | netfabric.ai

FMANO

Mon Sept 8 2025


Switzerland
Italy
Slovenia
Austria
Hungary
Croatia
Andorra
Tirana


Switzerland
Austria
Slovenia
Italy
Andorra
Prague
Vienna
Budapest
Zagreb
Sarajevo
Tirana
Podgorica

my.europe EUROPE NEWS

Flight delays after technical glitch closes Swiss airspace

COMMENTS

By **Euronews** with **AFP** • Updated: 15/06/2022

This network outage lasted

~2 hours

This network outage lasted

~2 hours

caused by a hardware failure

Facebook, Instagram, WhatsAp

nytimes.com/2021/10/04/technology/facebook-down.html

Update

TECHNOLOGY

The New York Times

LOG IN

Gone in Minutes, Out for Hours: Outage Shakes Facebook

When apps used by billions of people worldwide blinked out, lives were disrupted, businesses were cut off from customers — and some Facebook employees were locked out of their offices.

Give this article

885

This network outage lasted

~7 hours

This network outage lasted

~7 hours

caused by a misconfiguration

Networks are …

Networks are **easy** to break

Networks are **easy** to break

hard to repair

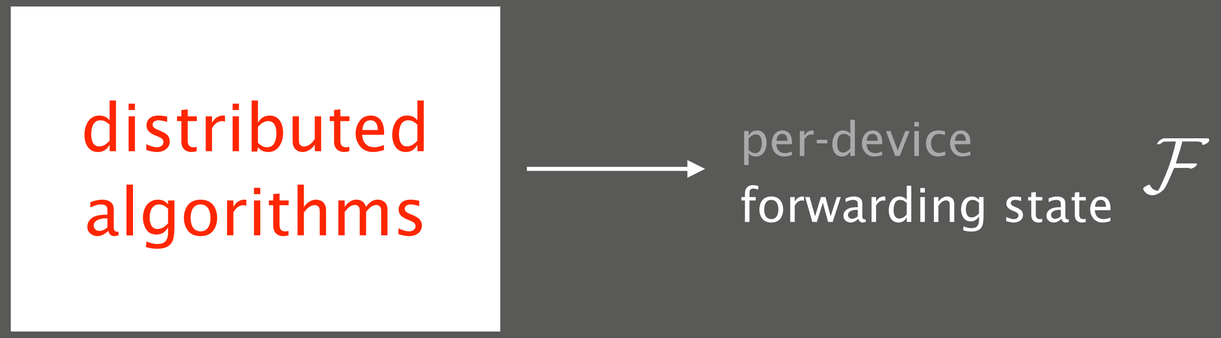
Networks are **easy** to break

hard to repair

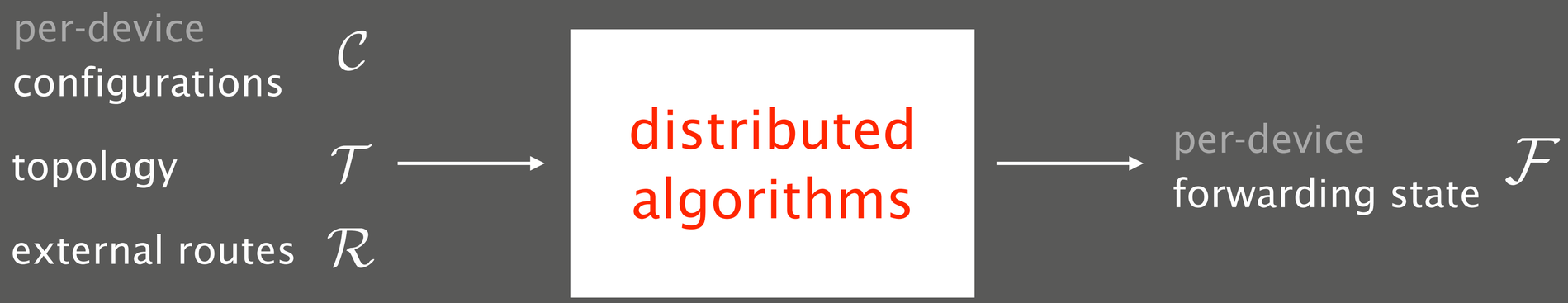
How come?

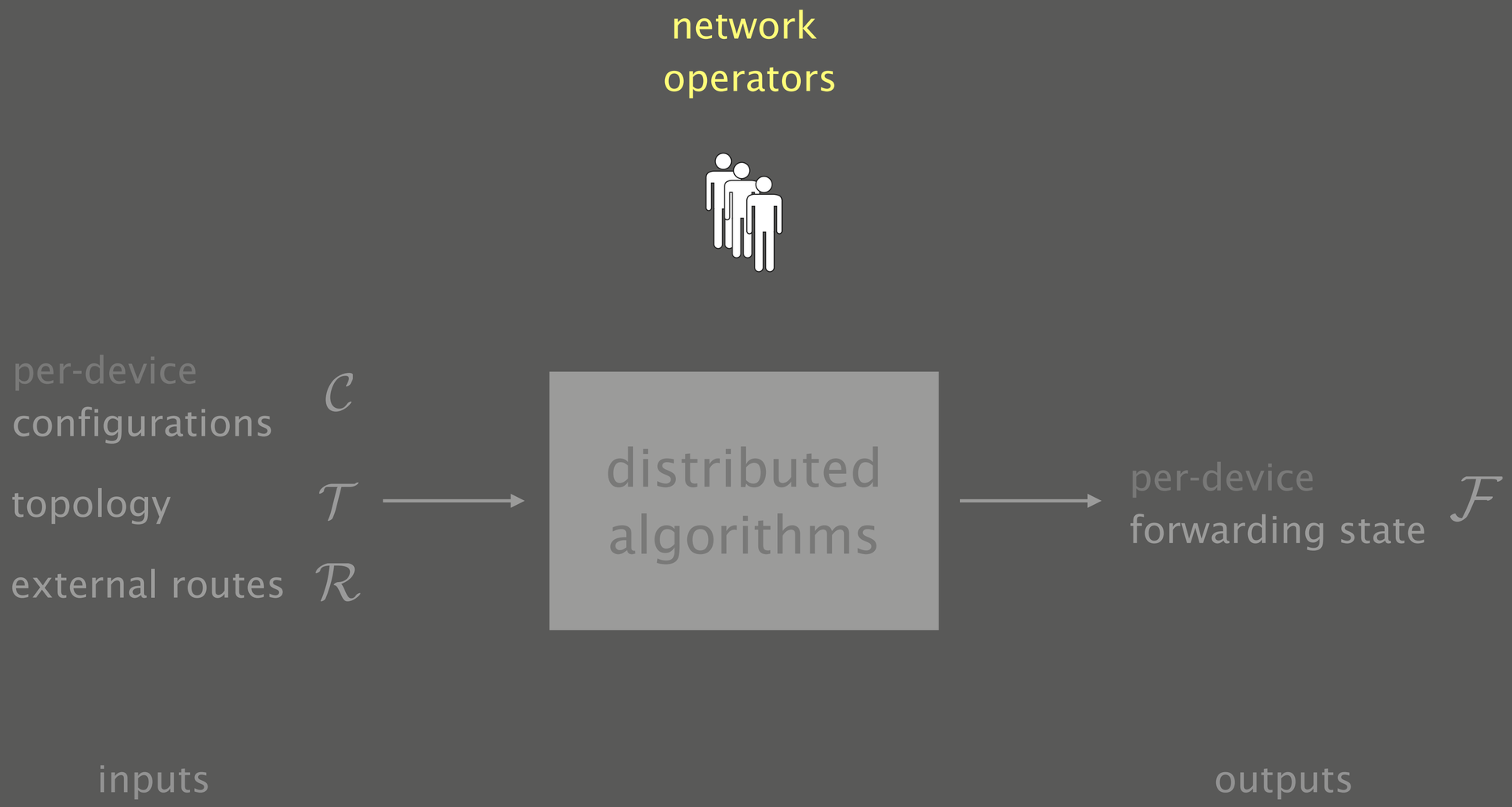
Configuring networks is hard because of
a fundamental semantic gap

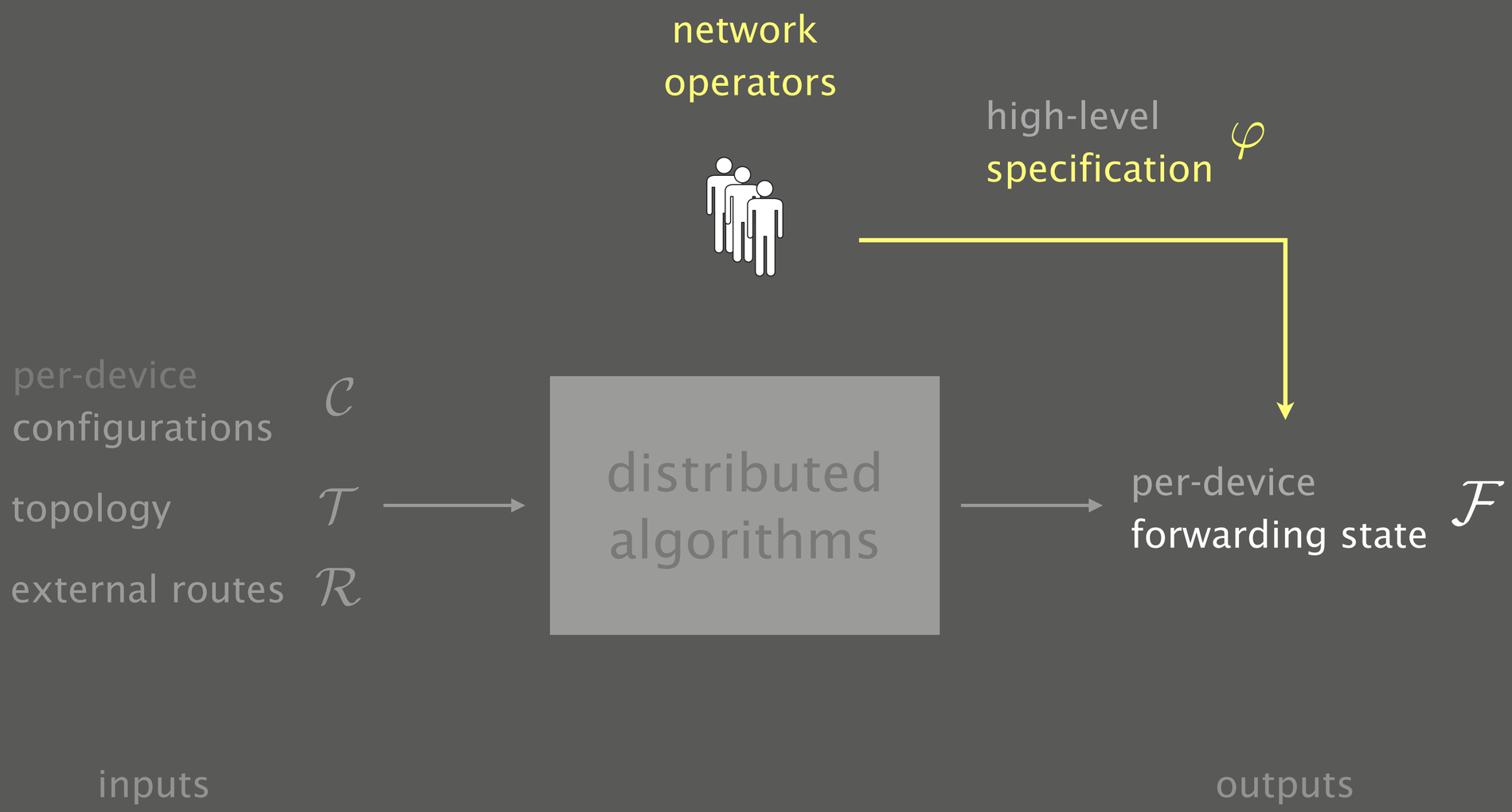
distributed
algorithms

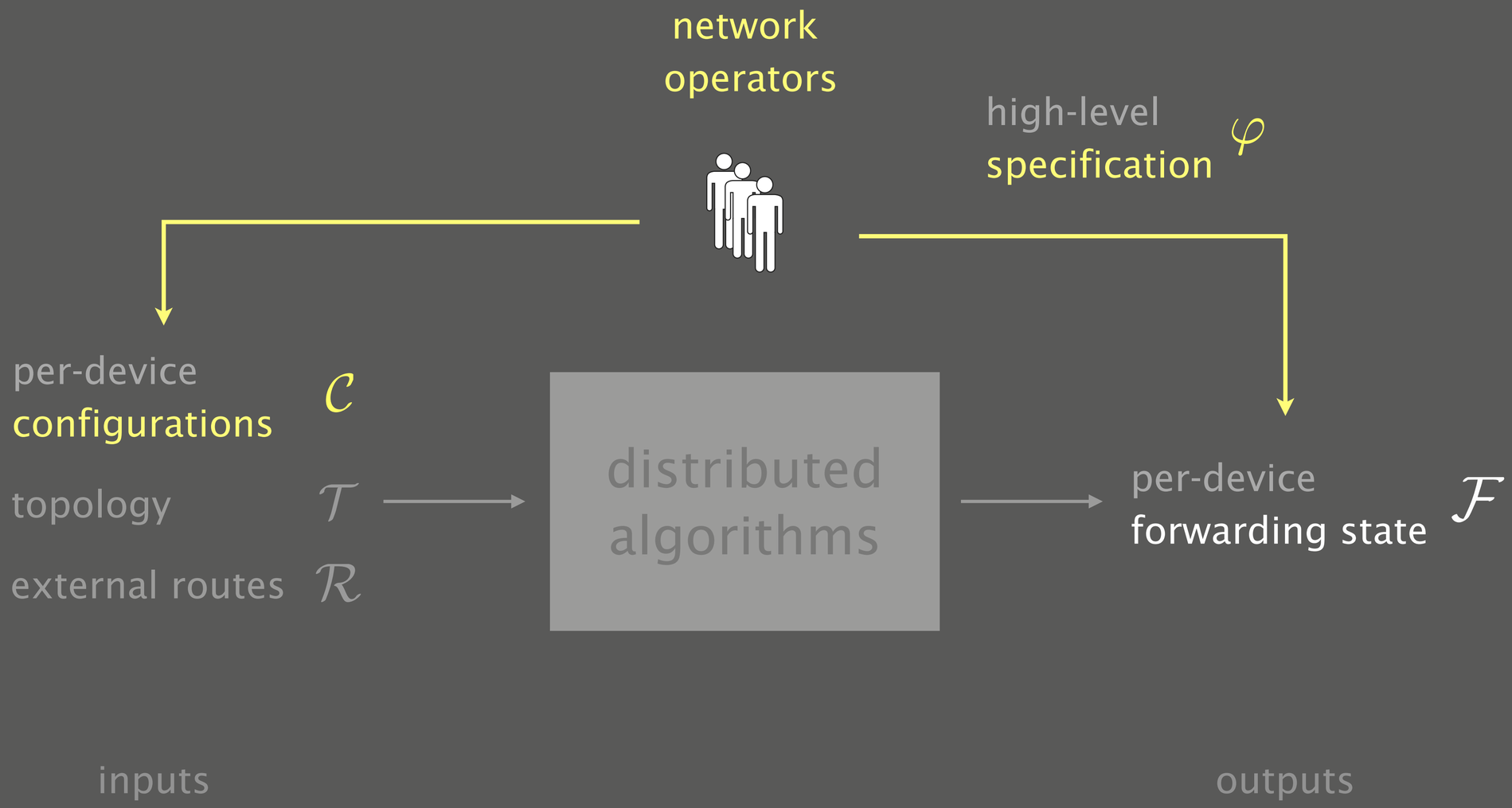

distributed
algorithms
per-device
forwarding state
$\mathcal{F}$

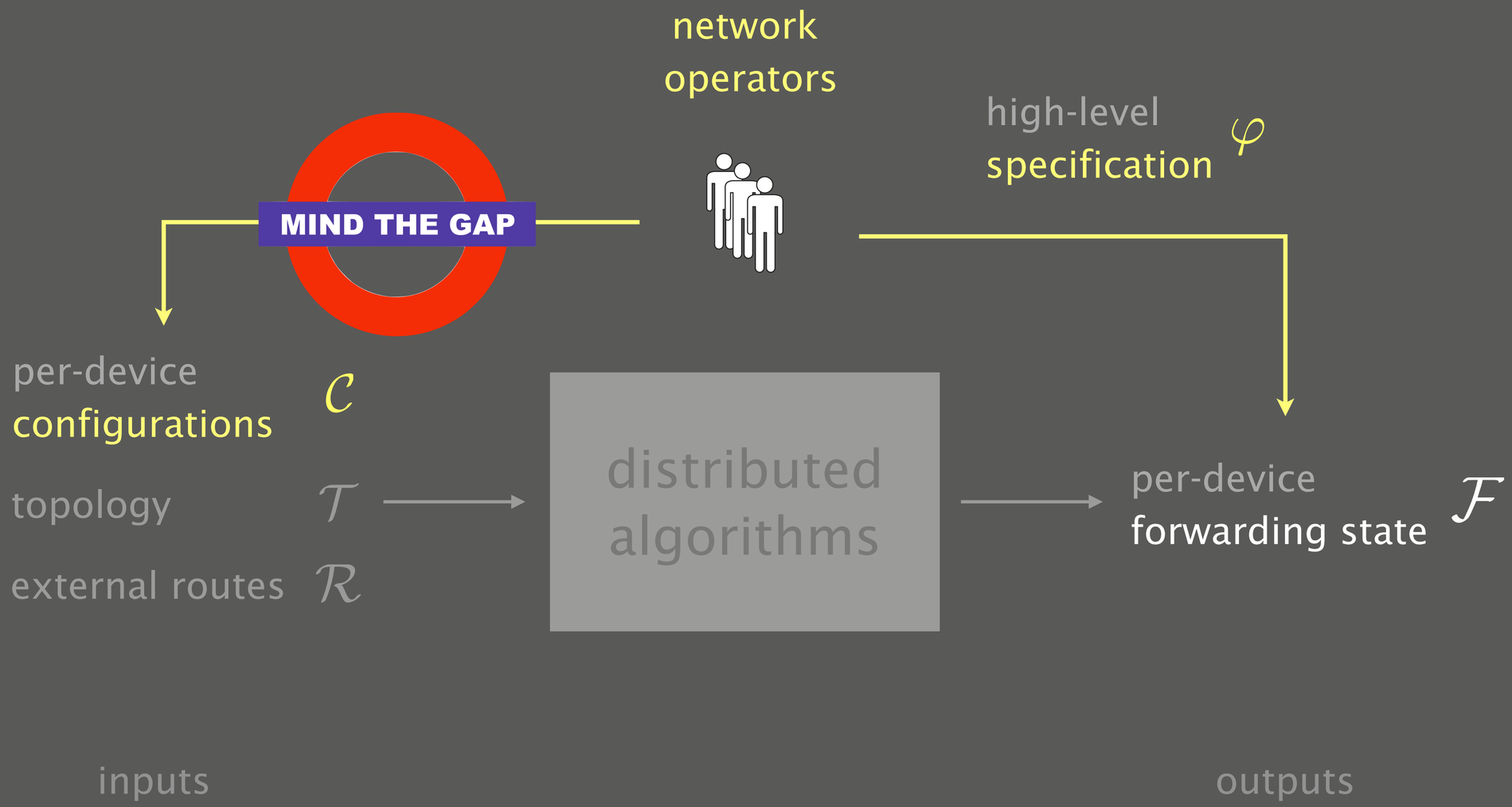
outputs


per-device
configurations
$\mathcal{C}$
topology
$\mathcal{T}$
external routes
$\mathcal{R}$
distributed
algorithms
per-device
forwarding state
$\mathcal{F}$
inputs
outputs


network
operators
per-device
configurations
$\mathcal{C}$
topology
$\mathcal{T}$
external routes
$\mathcal{R}$
distributed
algorithms
per-device
forwarding state
$\mathcal{F}$
inputs
outputs


network
operators
high-level
specification
φ
per-device
configurations
$\mathcal{C}$
topology
$\mathcal{T}$
external routes
$\mathcal{R}$
distributed
algorithms
per-device
forwarding state
$\mathcal{F}$
inputs
outputs


network
operators
high-level
specification
φ
per-device
configurations
$\mathcal{C}$
topology
$\mathcal{T}$
external routes
$\mathcal{R}$
distributed
algorithms
per-device
forwarding state
$\mathcal{F}$
inputs
outputs


network
operators
high-level
specification
φ
MIND THE GAP
per-device
configurations
$\mathcal{C}$
topology
$\mathcal{T}$
external routes
$\mathcal{R}$
distributed
algorithms
per-device
forwarding state
$\mathcal{F}$
inputs
outputs

Human factors contribute to
between 65% and 80% of all downtime incidents

Uptime Institute 2024 Outage Analysis

Network verification promises to significantly increase network reliability

Network verification promises to significantly increase network reliability

What is it?

Network verification

Given specification φ

Network verification

Given specification φ and configuration C

Network verification

Given specification φ and configuration C

Return $C \models \varphi$ ✓ ✗

Network verification has been wildly successful

Network verification has been wildly successful,

academically

Debugging the Data Plane with Anteater

Haohui Mai, Ahmed Khurshid, Rachit Agarwal
Matthew Caesar, P. Brighten Godfrey, Samuel T. King
University of Illinois at Urbana-Champaign
{mai4, khurshi1, agarwa16, caesar, pbg, kingst}@illinois.edu

ABSTRACT

Diagnosing problems in networks is a time-consuming and error-prone process. Existing tools to assist operators primarily focus on analyzing control plane configuration. Configuration analysis is limited in that it cannot find bugs in router software, and is harder to generalize across protocols since it must model complex configuration languages and dynamic protocol behavior.

This paper studies an alternate approach: diagnosing problems through static analysis of the data plane. This approach can catch bugs that are invisible at the level of configuration files, and simplifies unified analysis of a network across many protocols and implementations. We present *Anteater*, a tool for checking invariants in the data plane. Anteater translates high-level network invariants into instances of boolean satisfiability problems (SAT), checks them against network state using a SAT solver, and reports counterexamples if violations have been found. Applied to a large university network, Anteater revealed 23 bugs, including forwarding loops and stale ACL rules, with only five false positives. Nine of these faults are being fixed by campus network operators.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operation; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability

Keywords

Data Plane Analysis, Network Troubleshooting, Boolean Satisfiability

1. INTRODUCTION

Modern enterprise networks are complex, incorporating hundreds or thousands of network devices from multiple vendors performing diverse codependent functions such as routing, switching, and access control across physical and virtual networks (VPNs and VLANs). As in any complex computer system, enterprise networks are prone to a wide range of errors [10, 11, 12, 14, 25, 32, 38, 41], such as misconfiguration, software bugs, or unexpected interactions across protocols. These errors can lead to oscillations, black holes, faulty advertisements, or route leaks that ultimately cause disconnectivity and security vulnerabilities.

However, diagnosing problems in networks remains a black art. Operators often rely on heuristics — sending probes, reviewing logs, even observing mailing lists and making phone calls — that slow response to failures.[1] To address this, automated tools for network diagnostics [14, 43] analyze configuration files constructed by operators. While useful, these tools have two limitations stemming from their analysis *of* high-level configuration files. First, configuration analysis *cannot find bugs in router software*, which interprets and acts on those configuration files. Both commercial and open source router software regularly exhibit bugs that affect network availability or security [41] and have led to multiple high-profile outages and vulnerabilities [11, 44]. Second, configuration analysis *must model complex configuration languages and dynamic protocol behavior* in order to determine the ultimate effect of a configuration. As a result, these tools generally focus on checking correctness of a single protocol such as BGP [14, 15] or firewalls [2, 43]. Such diagnosis will be unable to reason about interactions that span multiple protocols, and may have difficulty dealing with the diversity in configuration languages from different vendors making up typical networks.

We take a different and complementary approach. Instead of diagnosing problems in the control plane, our goal is to *diagnose problems as close as possible to the network's actual behavior* through formal analysis of data plane state. Data plane analysis has two benefits. First, by checking the results of routing software rather than its inputs, we can catch bugs that are invisible at the level of configuration files, as with the Netcraft outage [11] mentioned above. Second, data plane analysis can encompass many different control plane implementations at once, including an entire network with routers from multiple vendors running different routing protocols.

[1]As one example, a Cisco design technote advises that "Unfortunately, there is no systematic procedure to troubleshoot an STA forwarding loop. ... Administrators generally do not have time to wait for the network to converge, so they must restore connectivity as soon as possible. The easy way out in this case is to manually disable every port that provides redundancy in the network. ... Each time you disable a port, check to see if you have restored connectivity in the network." [10]

SIGCOMM '11

Header Space Analysis: Static Checking For Networks

Peyman Kazemian, George Varghese, Nick McKeown
Stanford University, *UCSD and Yahoo! Research*, *Stanford University*
kazemian@stanford.edu, varghese@cs.ucsd.edu, nickm@stanford.edu

Abstract

Today's networks typically carry or deploy dozens of protocols and mechanisms simultaneously such as MPLS, NAT, ACLs and route redistribution. Even when individual protocols function correctly, failures can arise from the complex interactions of their aggregate, requiring network administrators to be masters of detail. Our goal is to automatically find an important class of failures, regardless of the protocols running, for both operational and experimental networks.

To this end we developed a general and protocol-agnostic framework, called *Header Space Analysis* (HSA). Our formalism allows us to statically check network specifications and configurations to identify an important class of failures such as *Reachability* Failures, *Forwarding Loops* and *Traffic Isolation and Leakage* problems. In HSA, protocol header fields are not first class entities; instead we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0,1\}^L$ space where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points (multicast).

We created a library of tools, called Hassel, to implement our framework, and used it to analyze a variety of networks and protocols. Hassel was used to analyze the Stanford University backbone network, and found all the forwarding loops in less than 10 minutes, and verified reachability constraints between two subnets in 13 seconds. It also found a large and complex loop in an experimental loose source routing protocol in 4 minutes.

1 Introduction

"Accidents will occur in the best-regulated families" — *Charles Dickens*

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was index into a forwarding table using a destination address, and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLAN and MPLS) appeared to escape from IP's limitations: e.g., NAT bypasses address limits and MPLS allows flexible routing. Further, new protocols for specific domains, such as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6,000 Internet RFCs and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously.

This complexity makes it daunting to operate a large network today. Network operators require great sophistication to master the complexity of many interacting protocols and middleboxes. The future is not any more rosy - complexity today makes operators wary of trying new protocols, even if they are available, for fear of breaking their network. Complexity also makes networks fragile, and susceptible to problems where hosts become isolated and unable to communicate. Debugging reachability problems is very time consuming. Even simple questions are hard to answer, such as *"Can Host A talk to Host B?"* or *"Can packets loop in my network?"* or *"Can User A listen to communications between Users B and C?"*. These questions are especially hard to answer in networks carrying multiple encapsulations and containing boxes that filter packets.

Thus, our first goal is to help system administrators statically analyze production networks today. We describe new methods and tools to provide formal answers to these questions, and many other failure conditions, *regardless of the protocols running in the network*.

Our second goal is to make it easier for system administrators to guarantee isolation between sets of hosts, users or traffic. Partitioning networks this way is usually called "slicing"; VLANs are a simple example used today. If configured correctly, we can be confident that traffic in one slice (e.g. a VLAN) cannot leak into another. This is useful for security, and to help answer questions such as *"Can I prevent Host A from talking to Host B?"*. For example, imagine two health-care providers using the same physical network. HIPAA [20] rules require that no information about a patient can be read by other providers. Thus a natural application of slicing is to place each provider in a separate slice and guarantee that no packet from one slice can be controlled by or read by the other slice. We call this *secure slicing*. Secure slicing may also be useful for banks as part of defense-in-depth, and for classified and unclassified users sharing the same physical network. Our tools can verify that slices have no physical links between them, and that packets injected at a port in one slice will only appear on ports in the same slice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) will ease the development of network applications, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [35].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [7] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [8, 18] is useful, but cannot find bugs in router software and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may not be true in an SDN network where controllers can be operated by other parties [15]. Another approach is to statically analyze snapshots of the network-wide data-plane state [5, 6, 11, 12]. These tools operate offline, and thus only find bugs after they happen.

This paper studies the following question: *Is it possible to check network-wide invariants, such as absence of routing loops, in real time as the network evolves?* This would enable us to check updates before they hit the network, allowing us to raise alarms, or even prevent bugs as they occur by blocking problematic changes. However, existing techniques for checking networks are not adequate for this purpose as they operate on timescales of seconds to hours [6, 11, 12].[1] As current SDN controllers are capable of handling around 30K new flow installs per second while maintaining a sub-10ms flow install time [16], rule verification latency in the order of seconds is not enough to ensure real-time response, and will affect controller throughput immensely. Delaying updates for processing can harm consistency of network state, and reduce reaction time of protocols with real-time requirements such as routing and fast failover. Moreover, checking network-wide properties seems to require network-wide state, and processing churn of a large network could introduce scaling challenges. Hence, we need some way to perform this checking at very high speeds.

We present a preliminary design, VeriFlow, which demonstrates that the goal of real-time verification is achievable.

[1]The average run time of reachability tests in [11] is 13 seconds.

NSDI '12

VeriFlow: Verifying Network-Wide Invariants in Real Time

Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, P. Brighten Godfrey
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, Illinois 61801-2302, USA
{khurshi1, wzhou10, caesar, pbg}@illinois.edu

ABSTRACT

Networks are complex and prone to bugs. Existing tools that check configuration files and data-plane state operate offline at timescales of seconds to hours, and cannot detect or prevent bugs as they arise.

Is it possible to *check network-wide invariants in real time*, as the network state evolves? The key challenge here is to achieve extremely low latency during the checks so that network performance is not affected. In this paper, we present a preliminary design, VeriFlow, which suggests that this goal is achievable. VeriFlow is a layer between a software-defined networking controller and network devices that checks for network-wide invariant violations dynamically as each forwarding rule is inserted. Based on an implementation using a Mininet OpenFlow network and Route Views trace data, we find that VeriFlow can perform rigorous checking within hundreds of microseconds per rule insertion.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management, Network monitoring*

General Terms

Algorithms, Design, Experimentation, Management, Performance, Security, Verification

Keywords

Software-defined networking, OpenFlow, forwarding, debugging, real time

1. INTRODUCTION

Network forwarding behaviors are complex, including codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. As a result, a substantial amount of effort is required to ensure networks' correctness and security. However, faults in the network state arise commonly in practice, including routing loops, misconfigured routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) will ease the development of network applications, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [35].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [7] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [8, 18] is useful, but cannot find bugs in router software and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may not be true in an SDN network where controllers can be operated by other parties [15]. Another approach is to statically analyze snapshots of the network-wide data-plane state [5, 6, 11, 12]. These tools operate offline, and thus only find bugs after they happen.

This paper studies the following question: *Is it possible to check network-wide invariants, such as absence of routing loops, in real time as the network evolves?* This would enable us to check updates before they hit the network, allowing us to raise alarms, or even prevent bugs as they occur by blocking problematic changes. However, existing techniques for checking networks are not adequate for this purpose as they operate on timescales of seconds to hours [6, 11, 12].[1] As current SDN controllers are capable of handling around 30K new flow installs per second while maintaining a sub-10ms flow install time [16], rule verification latency in the order of seconds is not enough to ensure real-time response, and will affect controller throughput immensely. Delaying updates for processing can harm consistency of network state, and reduce reaction time of protocols with real-time requirements such as routing and fast failover. Moreover, checking network-wide properties seems to require network-wide state, and processing churn of a large network could introduce scaling challenges. Hence, we need some way to perform this checking at very high speeds.

We present a preliminary design, VeriFlow, which demonstrates that the goal of real-time verification is achievable.

[1]The average run time of reachability tests in [11] is 13 seconds.

NSDI '13

A General Approach to Network Configuration Analysis

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan
Ramesh Govindan, Ratul Mahajan, Todd Millstein
University of California, Los Angeles, University of Southern California, Microsoft Research

Abstract— We present an approach to detect network configuration errors, which combines the benefits of two prior approaches. Like prior techniques that analyze configuration files, our approach can find errors proactively, before the configuration is applied, and answer "what if" questions. Like prior techniques that analyze data-plane snapshots, our approach can check a broad range of forwarding properties and produce actual packets that violate checked properties. We accomplish this combination by faithfully deriving and then analyzing the data plane that would emerge from the configuration. Our derivation of the data plane is fully declarative, employing a set of logical relations that represent the control plane, the data plane, and their relationship. Operators can query these relations to understand identified errors and their provenance. We use our approach to analyze two large university networks with qualitatively different routing designs and find many misconfigurations in each. Operators have confirmed the majority of these as errors and have fixed their configurations accordingly.

1 Introduction

Configuring networks is arduous because policy requirements (for resource management, access control, etc.) can be complex and configuration languages are low-level. Consequently, configurations errors that compromise availability, security, and performance are common [7, 21, 36]. In a recent incident, for example, a misconfiguration led to a nation-wide outage that impacted all customers of Time Warner for over an hour [3].

Prior approaches Researchers have developed two main approaches to detect network configuration errors. The first approach directly analyzes network configuration files [2, 5, 7, 24, 25, 28, 34]. Such a *static* analysis can flag errors proactively, before a new configuration is applied to the network, and it can naturally answer "what if" questions with respect to different environments (i.e., failures and route announcement from neighbors).

However, configurations of real networks are complex, with many interacting aspects (e.g., BGP, OSPF, ACLs, VLANs, static routing, route redistribution); existing configuration analysis tools handle this complexity by developing customized models for specific aspects of the configuration or specific correctness properties. For instance, rcc [7] produces a normalized representation of configuration that lets it check a range of properties that correspond to common errors (e.g., "route validity" of BGP, whether OSPF adjacencies are configured on both ends, and that there are no duplicate router identifiers). Similarly, FIREMAN [34] produces a "rule graph" structure to represent each ACL and analyzes these graphs. This selective focus makes configuration analysis practical, but it also limits the scope of what can be checked. Further, because many aspects of the configuration are not analyzed, it can be difficult for operators to assess how and whether identified errors ultimately impact forwarding.

Researchers have recently proposed a second approach that can be used to detect configuration errors: analyzing the data plane snapshots (i.e., forwarding behavior) of the network [13, 14, 22, 37]. Unlike with static analysis, any configuration error that causes undesirable forwarding can be precisely detected, because the data plane reflects the combined impact of all configuration aspects. Further, because the data plane has well-understood semantics and can be efficiently encoded in various logics, a wide range of forwarding properties can be concisely expressed and scalably checked with off-the-shelf constraint solvers.

Unfortunately, analysis of data plane snapshots cannot prevent errors proactively, before undesirable forwarding occurs. Further, once a problem is flagged, the operators still need to localize the responsible snippets of configuration. This task is challenging because the relationship between configuration snippets and forwarding behavior is complex. The responsible snippet is not necessarily

NSDI '15

A General Approach to Network Configuration Verification

Ryan Beckett, Aarti Gupta
Princeton University
Ratul Mahajan, David Walker
Microsoft Research & Intentionet

ABSTRACT

We present Minesweeper, a tool to verify that a network satisfies a wide range of intended properties such as reachability or isolation among nodes, waypointing, black holes, bounded path length, load-balancing, functional equivalence of two routers, and fault-tolerance. Minesweeper translates network configuration files into a logical formula that captures the stable states to which the network forwarding will converge as a result of interactions between routing protocols such as OSPF, BGP and static routes. It then combines the formula with constraints that describe the intended property. If the combined formula is satisfiable, there exists a stable state of the network in which the property does not hold. Otherwise, no stable state (if any) violates the property. We used Minesweeper to check four properties of 152 real networks from a large cloud provider. We found 120 violations, some of which are potentially serious security vulnerabilities. We also evaluated Minesweeper on synthetic benchmarks, and found that it can verify rich properties for networks with hundreds of routers in under five minutes. This performance is due to a suite of model-slicing and hoisting optimizations that we developed, which reduce runtime by over 460x for large networks.

CCS CONCEPTS

• **Networks** → *Network reliability;*

KEYWORDS

Network verification; Control plane analysis

ACM Reference format:
Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098834

1 INTRODUCTION

The control plane of traditional (non-SDN) networks is a complex distributed system. Network devices use one or more protocols to exchange information about topology and paths to various destinations. How they process this information and select paths to use for traffic depends on their local configuration files. These files tend to have thousands of lines of low-level directives, which makes it hard for humans to reason about them and even harder to reason about the network behavior that emerges through their interactions.

As a result, configuration errors that lead to costly outages are all too common. Indeed, every few months configuration-induced outages at major networks make the news [1, 5, 29, 32]. Systematic surveys also show that configuration error is the biggest contributor to such network outages [20, 26].

To address this problem, researchers have developed many tools for finding errors in network configurations. We broadly classify these tools along two dimensions: (*i*) *network design coverage*—types of network topologies, routing protocols and other features the tool supports; and (*i*) *data plane coverage*—how many (or how much) of the possible data planes the tool can analyze. The network control plane dynamically generates different data planes as its environment (i.e., up/down status of links and routing announcements received from external neighbors) changes. Tools with higher data plane coverage can analyze more such data planes.

Some of the earliest network diagnostic tools such as *traceroute* and *ping* can help find configuration errors by analyzing whether and how a given packet reaches its destination. These tools are simple and have high network design coverage—they can analyze forwarding for any network topology or routing protocol. But they have poor data plane coverage—for each run, they analyze the forwarding behavior for only a single packet for the data plane that is currently installed in the network.

A more recent class of data *plane analysis* tools such as HSA [18] and Veriflow [19] have better data plane coverage. They can analyze reachability for all packets between two machines, rather than just one packet. However, the data plane coverage of such tools is still far less than ideal because they analyze only the data plane that is currently installed in the network. That is, they can only find errors *after* the network has produced the erroneous data plane.

Control plane analysis tools such as Batfish [15] can find configuration errors proactively, before deploying potentially buggy configurations. Batfish takes the network configuration (*i.e.*, its control plane) and a specific environment (*e.g.*, a link-failure scenario) as input and analyzes the resulting data plane. This ability allows operators to go beyond the current data plane and analyze future data planes that may arise under different environments. Still, each run of Batfish allows users to explore at most one data plane, and given the large number of possible environments, it is intractable to guarantee correctness for all possible data planes.

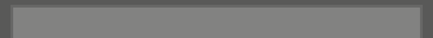
Must recently, several control plane analysis tools have gone from testing individual data planes to verification—that is, reasoning

SIGCOMM '17

and many, *many* more

Network verification has been wildly successful,

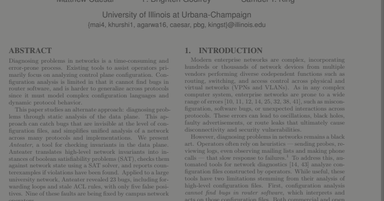
academically *and* commercially



Debugging the Data Plane with Anteater

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, Samuel T. King

University of Illinois at Urbana-Champaign
{mai4, khurshi1, agarwa16, caesar, pbg, kingst}@illinois.edu



ABSTRACT

Diagnosing problems in networks is a time-consuming and error-prone process. Existing tools to assist operators primarily focus on analyzing control plane configuration. Configuration analysis is limited in that it cannot find bugs in router software, and is harder to generalize across protocols since it must model complex configuration languages and dynamic protocol behavior.

This paper studies an alternate approach: diagnosing problems through static analysis of the data plane. This approach can catch bugs that are invisible at the level of configuration files, and simplifies unified analysis of a network across many protocols and implementations. We present *Anteater*, a tool for checking invariants in the data plane. Anteater translates high-level network invariants into instances of boolean satisfiability problems (SAT), checks them against network state using a SAT solver, and reports counterexamples if violations have been found. Applied to a large university network, Anteater revealed 23 bugs, including forwarding loops and stale ACL rules, with only five false positives. Nine of these faults are being fixed by campus network operators.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operation; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability

Keywords

Data Plane Analysis, Network Troubleshooting, Boolean Satisfiability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'11, August 15-19, 2011, Toronto, Ontario, Canada.
Copyright 2011 ACM 978-1-4503-0797-0/11/08 ...$10.00.

1. INTRODUCTION

Modern enterprise networks are complex, incorporating hundreds or thousands of network devices from multiple vendors performing diverse independent functions such as routing, switching, and access control across physical and virtual networks (VPNs and VLANs). As in any complex computer system, enterprise networks are prone to a wide range of errors [10, 11, 12, 14, 25, 32, 38, 41], such as misconfiguration, software bugs, or unexpected interactions across protocols. These errors can lead to oscillations, black holes, faulty advertisements, or route leaks that ultimately cause disconnectivity and security vulnerabilities.

However, diagnosing problems in networks remains a black art. Operators often rely on heuristics – sending probes, inspecting logs, even observing mailing lists and making phone calls – that show response to failures.[1] To address this, automated tools for network diagnostics [14, 43] analyze configuration files constructed by operators. While useful, these tools have two limitations stemming from their analysis of high-level configuration files. First, configuration analysis cannot *find bugs in router software*, which interprets and acts on those configuration files. Both commercial and open source router software regularly exhibit bugs that affect network availability or security [41] and have led to multiple high-profile outages and vulnerabilities [11, 44]. Second, configuration analysis *must model complex configuration languages and dynamic protocol behavior* in order to determine the ultimate effect of a configuration. As a result, these tools generally focus on checking correctness of a single protocol such as BGP [14, 15] or firewalls [2, 43]. Such diagnosis will be unable to reason about interactions that span multiple protocols, and may have difficulty dealing with the diversity in configuration languages from different vendors making up typical networks.

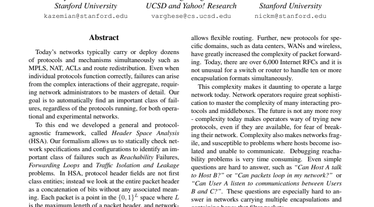
We take a different and complementary approach. Instead of diagnosing problems in the control plane, our goal is to *diagnose problems as close as possible to the network's actual behavior* through formal analysis of data plane state. Data plane analysis has two benefits. First, by checking the results of routing software rather than its inputs, we can catch bugs that are invisible at the level of configuration

[1]As one example, a Cisco design technote advises that "Unfortunately, there is no mechanism currently that can show how an IGP has converged throughout the entire good/bad/ugly network. Instead, the easiest way to look for the cause of the issue is to start at one end of the network and work as quickly as possible. The easy way out in this case is to manually disable every port that provides redundancy in the network. ... Each time you disable a port, check to see if you have restored connectivity in the network." [10]

290

Header Space Analysis: Static Checking For Networks

Peyman Kazemian, *Stanford University*, kazemian@stanford.edu
George Varghese, *UCSD and Yahoo! Research*, varghese@cs.ucsd.edu
Nick McKeown, *Stanford University*, nickm@stanford.edu



Abstract

Today's networks typically carry or deploy dozens of protocols and mechanisms simultaneously such as MPLS, NAT, ACLs and route redistribution. Even when individual protocols function correctly, failures can arise from the complex interactions of their aggregate, requiring network administrators to be masters of detail. Our goal is to automatically find an important class of failures, regardless of the protocols running, for both operational and experimental networks.

To this end we developed a general and protocol-agnostic framework, called *Header Space Analysis* (HSA). Our formalism allows us to statically check network specifications and configurations to identify an important class of failures such as *Reachability* Failures, *Forwarding Loops* and *Traffic Isolation and Leakage* problems. In HSA, protocol header fields are not first class entities; instead we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0,1\}^L$ space where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points (multicast).

We created a library of tools, called Hassel, to implement our framework, and used it to analyze a variety of networks and protocols. Hassel was used to analyze the Stanford University backbone network, and found all the forwarding loops in less than 10 minutes, and verified reachability constraints between two subsets in 13 seconds. It also found a large and complex loop in an experimental loose source routing protocol in 4 minutes.

1 Introduction

"Accidents will occur in the best-regulated families" — *Charles Dickens*

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was index into a forwarding table using a destination address, and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLAN and MPLS) appeared to escape from IP's limitations: e.g., NAT bypasses address limits and MPLS allows flexible routing. Further, new protocols for specific domains, such as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6,000 Internet RFCs and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously.

This complexity makes it daunting to operate a large network today. Network operators require great sophistication to master the complexity of many interacting protocols and middleboxes. The future is not any more rosy - complexity today makes operators wary of trying new protocols, even if they are available, for fear of breaking their network. Complexity also makes networks fragile, and susceptible to problems where hosts become isolated and unable to communicate. Debugging reachability problems is very time consuming. Even simple questions are hard to answer, such as *"Can Host A talk to Host B?"* or *"Can packets loop in my network?"* or *"Can User A listen to communications between Users B and C?"*. These questions are especially hard to answer in networks carrying multiple encapsulations and containing boxes that filter packets.

Thus, our first goal is to help system administrators statically analyze production networks today. We ascribe new methods and tools to provide formal answers to these questions, and many other failure conditions, *regardless of the protocols running in the network*.

Our second goal is to make it easier for system administrators to guarantee isolation between sets of hosts, users or traffic. Partitioning networks this way is usually called "slicing"; VLANs are a simple example used today. If configured correctly, we can be confident that traffic in one slice (e.g. a VLAN) cannot leak into another. This is useful for security, and to help answer questions such as *"Can I prevent Host A from talking to Host B?"*. For example, imagine two health-care providers using the same physical network. HIPAA [20] rules require that no information about a patient can be read by other providers. Thus a natural application of slicing is to place each provider in a separate slice and guarantee that no packet from one slice can be controlled by or read by the other slice. We call this *secure slicing*. Secure slicing may also be useful for banks as part of defense-in-depth, and for classified and unclassified users sharing the same physical network. Our tools can verify that slices have

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2012 ACM 978-1-4503-1477-0/12/08 ...$15.00.

49

VeriFlow: Verifying Network-Wide Invariants in Real Time

Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, P. Brighten Godfrey
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, Illinois 61801-2302, USA
{khurshi1, wzhou10, caesar, pbg}@illinois.edu

ABSTRACT

Networks are complex and prone to bugs. Existing tools that check configuration files and data-plane state operate offline at timescales of seconds to hours, and cannot detect or prevent bugs as they arise.

Is it possible to *check network-wide invariants in real time*, as the network state evolves? The key challenge here is to achieve extremely low latency during the checks so that network performance is not affected. In this paper, we present a preliminary design, VeriFlow, which suggests that this goal is achievable. VeriFlow is a layer between a software-defined networking controller and network devices that checks for network-wide invariant violations dynamically as each forwarding rule is inserted. Based on an implementation using a Mininet OpenFlow network and Route Views trace data, we find that VeriFlow can perform rigorous checking within hundreds of microseconds per rule insertion.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management, Network monitoring*

General Terms

Algorithms, Design, Experimentation, Management, Performance, Security, Verification

Keywords

Software-defined networking, OpenFlow, forwarding, debugging, real time

1. INTRODUCTION

Network forwarding behaviors are complex, including codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. As a result, a substantial amount of effort is required to ensure networks' correctness and security. However, faults in the network state arise commonly in practice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) will ease the development of network applications, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [35].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [7] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [8, 18] is useful, but cannot find bugs in router software and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may not be true in an SDN network where controllers can be operated by other parties [15]. Another approach is to statically analyze snapshots of the network-wide data-plane state [5, 6, 11, 12]. These tools operate offline, and thus only find bugs after they happen.

This paper studies the following question: *Is it possible to check network-wide invariants, such as absence of routing loops, in real time as the network evolves?* This would enable us to check updates before they hit the network, allowing us to raise alarms, or even prevent bugs as they occur by blocking problematic changes. However, existing techniques for checking networks are not adequate for this purpose as they operate on timescales of seconds to hours [6, 11, 12].[1] As current SDN controllers are capable of handling around 30K new flow installs per second while maintaining a sub-10ms flow install time [16], rule verification latency in the order of seconds is not enough to ensure real-time response, and will affect controller throughput immensely. Delaying updates for processing can harm consistency of network state, and reduce reaction time of protocols with real-time requirements such as routing and fast failover. Moreover, checking network-wide properties seems to require network-wide state, and processing churn of a large network could introduce scaling challenges. Hence, we need some way to perform this checking at very high speeds.

We present a preliminary design, VeriFlow, which demonstrates that the goal of real-time verification is achievable.

[1]The average run time of reachability tests in [11] is 13 seconds.

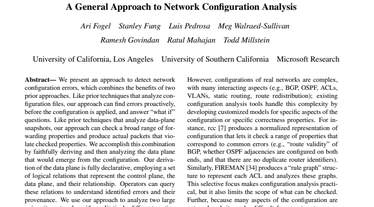
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HotSDN'12, August 13, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1477-0/12/08 ...$15.00.

49

A General Approach to Network Configuration Analysis

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, Todd Millstein

University of California, Los Angeles; University of Southern California; Microsoft Research



Abstract— We present an approach to detect network configuration errors, which combines the benefits of two prior approaches. Like prior techniques that analyze configuration files, our approach can find errors proactively, before the configuration is applied, and answer "what if" questions. Like prior techniques that analyze data-plane snapshots, our approach can check a broad range of forwarding properties and produce actual packets that violate checked properties. We accomplish this combination by faithfully deriving and then analyzing the data plane that would emerge from the configuration. Our derivation of the data plane is fully declarative, employing a set of logical relations that represent the control plane, the data plane, and their relationship. Operators can query these relations to understand identified errors and their provenance. We use our approach to analyze two large university networks with qualitatively different routing designs and find many misconfigurations in each. Operators have confirmed the majority of these as errors and have fixed their configurations accordingly.

1 Introduction

Configuring networks is arduous because policy requirements (for resource management, access control, etc.) can be complex and configuration languages are low-level. Consequently, configurations errors that compromise availability, security, and performance are common [7, 21, 36]. In a recent incident, for example, a misconfiguration led to a nation-wide outage that impacted all customers of Time Warner for over an hour [3].

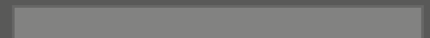
Prior approaches Researchers have developed two main approaches to detect network configuration errors. The first approach directly analyzes network configuration files [2, 5, 7, 24, 25, 28, 34]. Such a *static* analysis can flag errors proactively, before a new configuration is applied to the network, and it can naturally answer "what if" questions with respect to different environments (i.e., failures and route announcement from neighbors).

However, configurations of real networks are complex, with many interacting aspects (e.g., BGP, OSPF, ACLs, VLANs, static routing, route redistribution); existing configuration analysis tools handle this complexity by developing customized models for specific aspects of the configuration or specific correctness properties. For instance, rcc [7] produces a normalized representation of configuration that lets it check a range of properties that correspond to common errors (e.g., "route validity" of BGP, whether OSPF adjacencies are configured on both ends, and that there are no duplicate router identifiers). Similarly, FIREMAN [34] produces a "rule graph" structure to represent each ACL and analyzes these graphs. This selective focus makes configuration analysis practical, but it also limits the scope of what can be checked. Further, because many aspects of the configuration are not analyzed, it can be difficult for operators to assess how and whether identified errors ultimately impact forwarding.

Researchers have recently proposed a second approach that can be used to detect configuration errors: analyzing the data plane snapshots (i.e., forwarding behavior) of the network [13, 14, 22, 37]. Unlike with static analysis, any configuration error that causes undesirable forwarding can be precisely detected, because the data plane reflects the combined impact of all configuration aspects. Further, because the data plane has well-understood semantics and can be efficiently encoded in various logics, a wide range of forwarding properties can be concisely expressed and scalably checked with off-the-shelf constraint solvers.

Unfortunately, analysis of data plane snapshots cannot prevent errors proactively, before undesirable forwarding occurs. Further, once a problem is flagged, the operators still need to localize the responsible snippets of configuration. This task is challenging because the relationship between configuration snippets and forwarding behavior is complex. The responsible snippet is not necessarily

USENIX Association 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) 469



A General Approach to Network Configuration Verification

Ryan Beckett, Princeton University
Aarti Gupta, Princeton University
Ratul Mahajan, Microsoft Research & Intentionet
David Walker, Princeton University

ABSTRACT

We present Minesweeper, a tool to verify that a network satisfies a wide range of intended properties such as reachability or isolation among nodes, waypointing, black holes, bounded path length, load-balancing, functional equivalence of two routers, and fault-tolerance. Minesweeper translates network configurations files into a logical formula that captures the stable states to which the network forwarding will converge as a result of interactions between routing protocols such as OSPF, BGP and static routes. It then combines the formula with constraints that describe the intended property. If the combined formula is satisfiable, there exists a stable state of the network in which the property does not hold. Otherwise, no stable state (if any) violates the property. We used Minesweeper to check four properties of 152 real networks from a large cloud provider. We found 120 violations, some of which are potentially serious security vulnerabilities. We also evaluated Minesweeper on synthetic benchmarks, and found that it can verify properties for networks with hundreds of routers in under five minutes. This performance is due to a suite of model-slicing and hoisting optimizations that we developed, which reduce runtime by over 460x for large networks.

Some of the earliest network diagnostic tools such as traceroute and ping can help find configuration errors by analyzing whether and how a given packet reaches its destination. These tools are simple and have high network design coverage—they can analyze forwarding for any network topology or routing protocol. But they have poor data plane coverage—for each run, they analyze the forwarding behavior for only a single packet for the data plane that is currently installed in the network.

A more recent class of data plane analysis tools such as HSA [16] and Veriflow [19] have better data plane coverage. They can analyze reachability for all packets between two machines, rather than just one packet. However, the data plane coverage of such tools is still far less than ideal because they analyze only the data plane that is currently installed in the network. That is, they can only find errors *after* the network has produced the erroneous data plane.

Control plane analysis tools such as Batfish [13] can find configuration errors proactively, before deploying potentially buggy configurations. Batfish takes the network configuration (i.e., its control plane) and a specific environment (e.g., a link-failure scenario) as input and analyzes the resulting data plane. This ability allows operators to go beyond the current data plane and analyze future data planes that may arise under different environments. Still, each run of Batfish allows users to explore at most one data plane, and given the large number of possible environments, it is intractable to guarantee correctness for all possible data planes.

Most recently, several control plane analysis tools have gone from testing individual data planes to verification—that is, reasoning

exchange information about topology and paths to various destinations. How they process this information and select paths to use for traffic depends on their local configuration files. These files tend to have thousands of lines of low-level directives, which makes it hard for humans to reason about them and even harder to reason about the network behavior that emerges through their interactions.

As a result, configuration errors that lead to costly outages are all too common. Indeed, every few months configuration-induced outages at major networks make the news [1, 3, 29, 32]. Systematic surveys also show that configuration error is the biggest contributor to such network outages [20, 26].

To address this problem, researchers have developed many tools for finding errors in network configurations. We broadly classify these tools along two dimensions: i) *network design coverage*—types of network topologies, routing protocols and other features the tool supports; and ii) *data plane coverage*—how many (or how much) of the possible data planes the tool can analyze. The network control plane dynamically generates different data planes as its environment (i.e., up/down status of links and routing announcements received from external neighbors) changes. Tools with higher data plane coverage can analyze more such data planes.

CCS CONCEPTS

• Networks → *Network reliability*;

KEYWORDS

Network verification; Control plane analysis

ACM Reference format:
Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098854

1 INTRODUCTION

The control plane of traditional (non-SDN) networks is a complex, distributed system. Network devices use one or more protocols to

155

Forward Networks

Veriflow

acquired by VMware

Intentionet

acquired by Amazon

and, yet...

and, yet…

networks *still* go down

due to misconfiguration

Either network operators…

Either network operators

use verification…

Either network operators

use verification

but it is not enough or not "in the right way"

Either network operators

use verification
but it is not enough or not "in the right way"

do not use verification...

Either network operators

use verification
but it is not enough or not "in the right way"

do not use verification…
voluntarily or not

Either network operators

use verification
but it is not enough or not "in the right way"

do not use verification
voluntarily or not

What's missing?

Network verification ain't a panacea

Network verification ain't a panacea

Network verification…

checks a restricted set of properties

and misses out on important ones

can be imprecise

over-/under-approximates behaviors

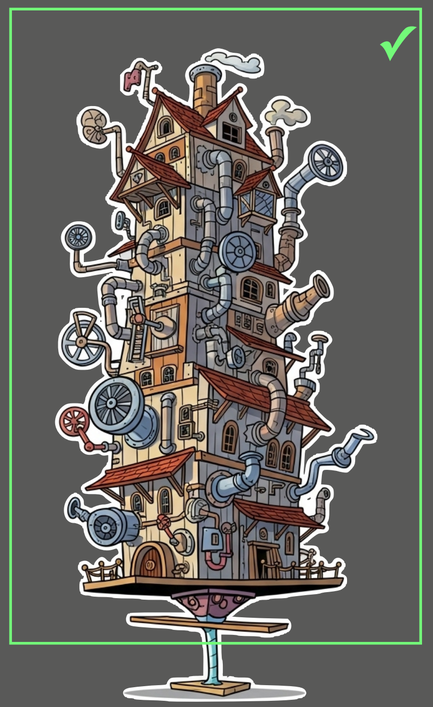
is not user-friendly

a perennial problem in formal methods

Verifying configurations was the easy part

My ongoing quest toward correct network operations

Formally Verified



More properties

beyond reachability

More coverage

beyond handcrafted models

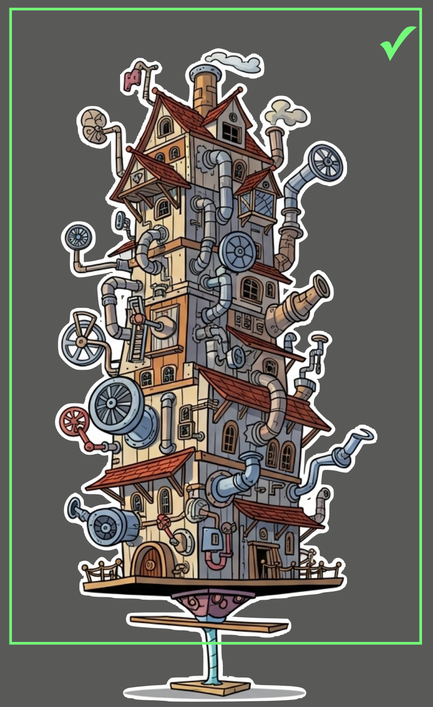
Better usability

beyond human intervention

Verifying configurations was the easy part

My ongoing quest toward correct network operations

Formally Verified



More properties

beyond reachability

More coverage

beyond handcrafted models

Better usability

beyond human intervention

Existing verifiers typically only check forwarding-level properties, in the steady state

Existing verifiers typically only check

forwarding-level properties, in the steady state

reachability, isolation,
specific path(s) being followed…

Existing verifiers typically only check forwarding-level properties, in the steady state

once the network has converged,
and assuming it converges

Operators also often care about other types of properties,

as well as how long and how often these are violated

NSDI 2025

Verifying maximum link loads in a changing world

Tibor Schneider
ETH Zürich

Stefano Vissicchio
University College London

Laurent Vanbever
ETH Zürich

Abstract

To meet ever more stringent requirements, network operators often need to reason about worst-case link loads. Doing so involves analyzing traffic forwarding after failures and BGP route changes. State-of-the-art systems identify failure scenarios causing congestion, but they ignore route changes.

We present Velo, the first verification system that efficiently finds maximum link loads under failures *and* route changes. The key building block of Velo is its ability to massively reduce the gigantic space of possible route changes thanks to (i) a router-based abstraction for route changes, (ii) a theoretical characterization of scenarios leading to worst-case link loads, and (iii) an approximation of input traffic matrices. We fully implement and extensively evaluate Velo. Velo takes only a few minutes to accurately compute all worst-case link loads in large ISP networks. It thus provides operators with critical support to robustify network configurations, improve network management and take business decisions.

1 Introduction

Link loads—how much traffic crosses each link—are a key indicator of network performance. High link loads, in partic-

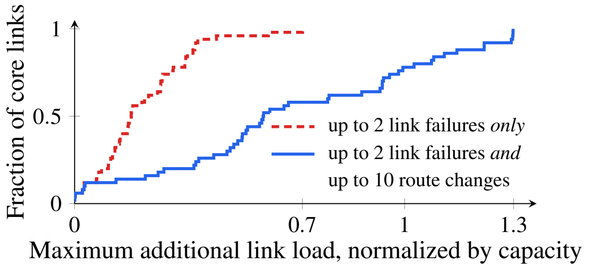
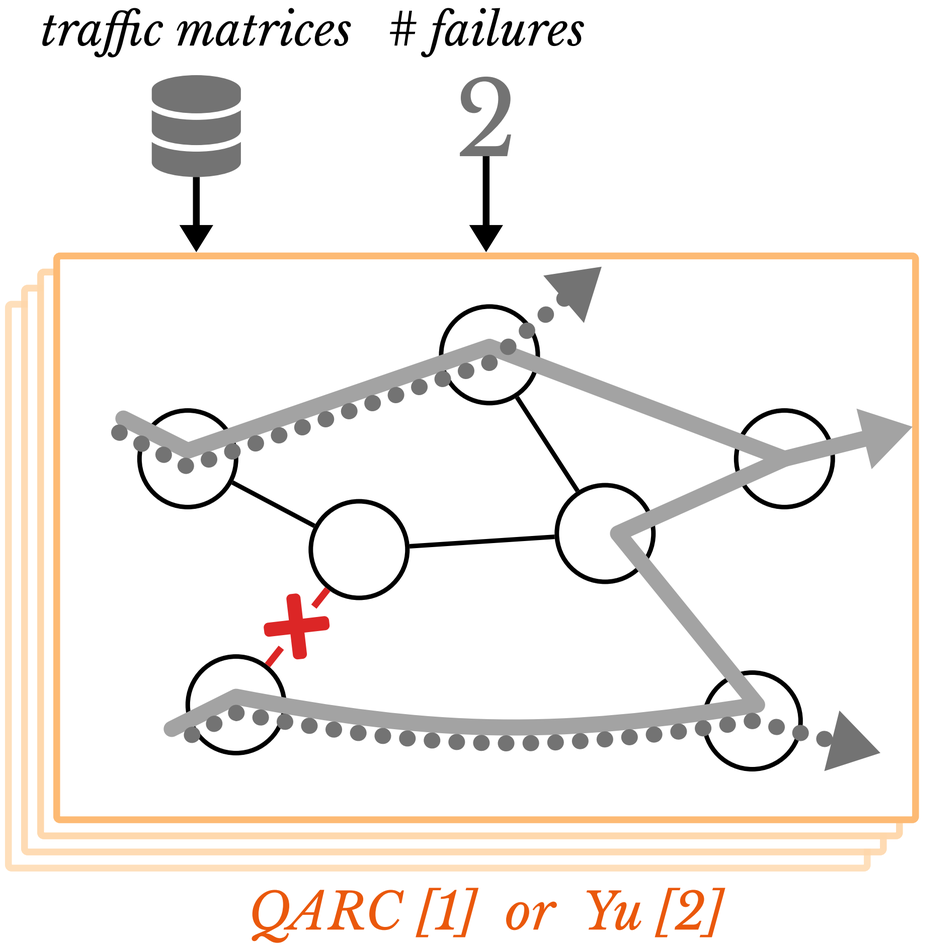


Figure 1: Additional link loads can double due to route changes compared with link failures only. This plot shows the additional traffic on all the core links of an ISP network.

Reasoning about link loads is beyond the capabilities of existing network verifiers. Most of them [1, 4, 14, 39, 40, 42, 45, 50] do not support *performance* requirements such as maximum link loads, but restrict to *functional* requirements, such as the absence of blackholes and forwarding loops.

A couple of recent contributions [7, 27, 44] focus on assessing properties on link loads. Yet, they only consider network failures and assume fixed external routes, fitting controlled network environments such as data centers. For most Internet-connected networks, *both* failures and route changes

Recent systems find worst-case link loads under failures.

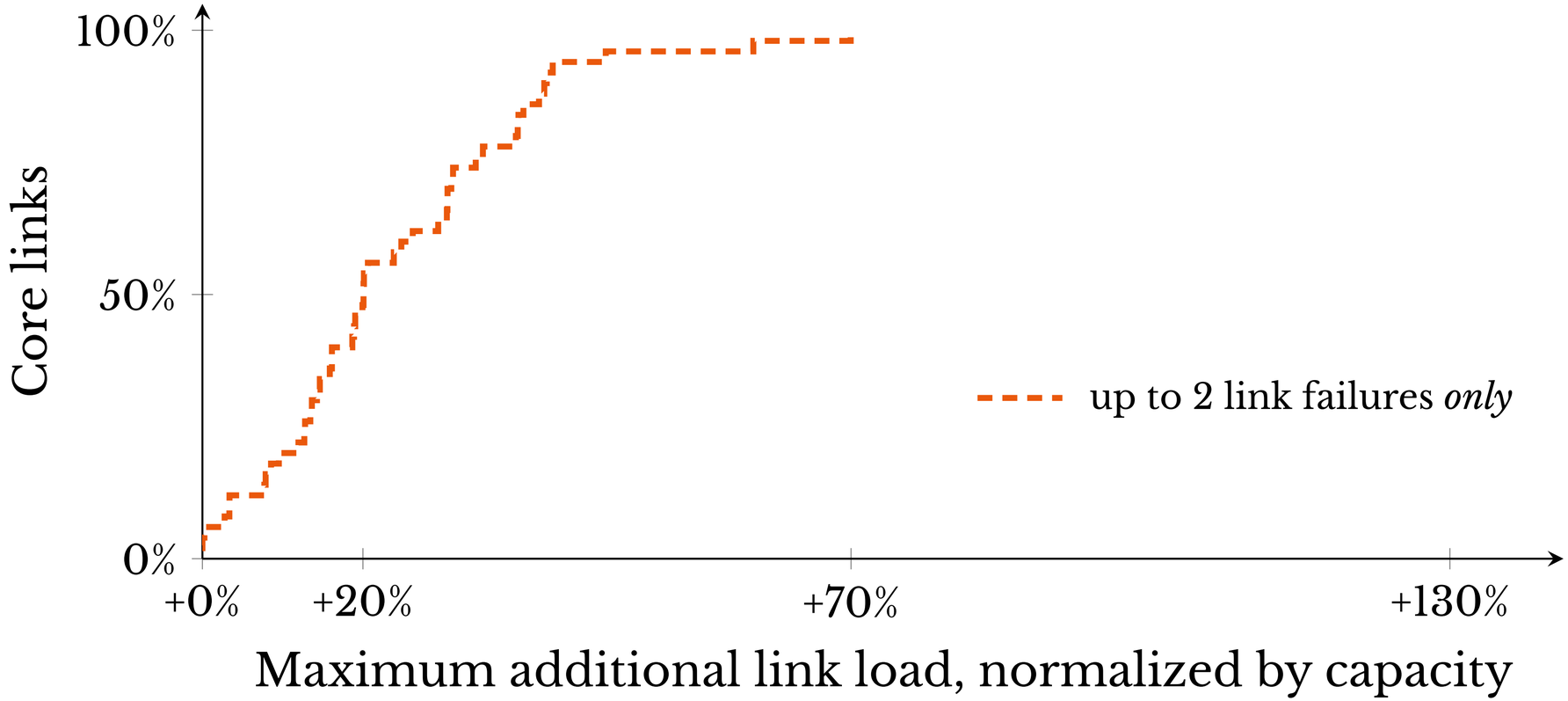


Find *worst-case* loads under arbitrary failures.

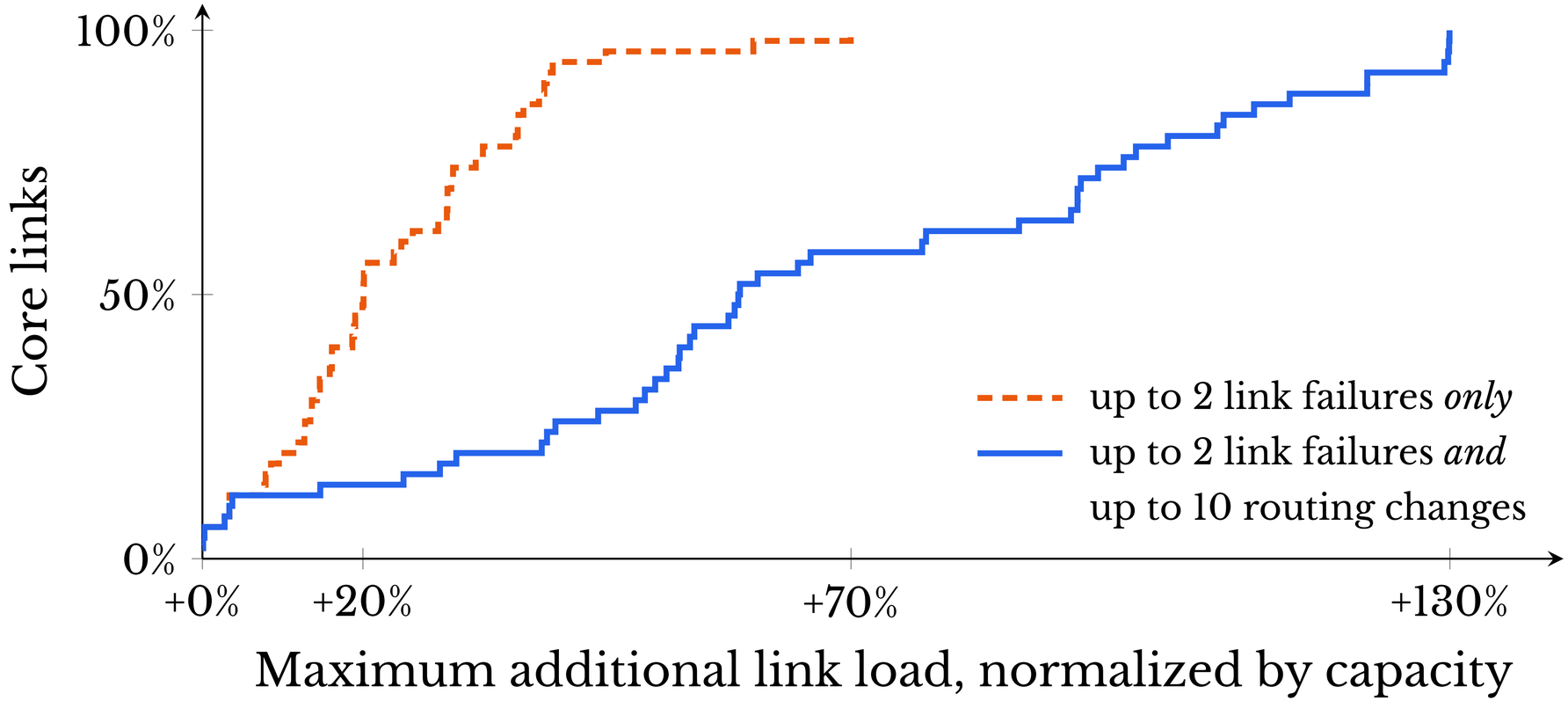
- Traffic also depends on *BGP routing inputs.*

[1] Kausik Subramanian et al. “Detecting network load violations for distributed control planes”. In: *ACM SIGPLAN*. 2020
[2] Ruihan Li et al. “A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures”. In: *ACM SIGCOMM*. 2024

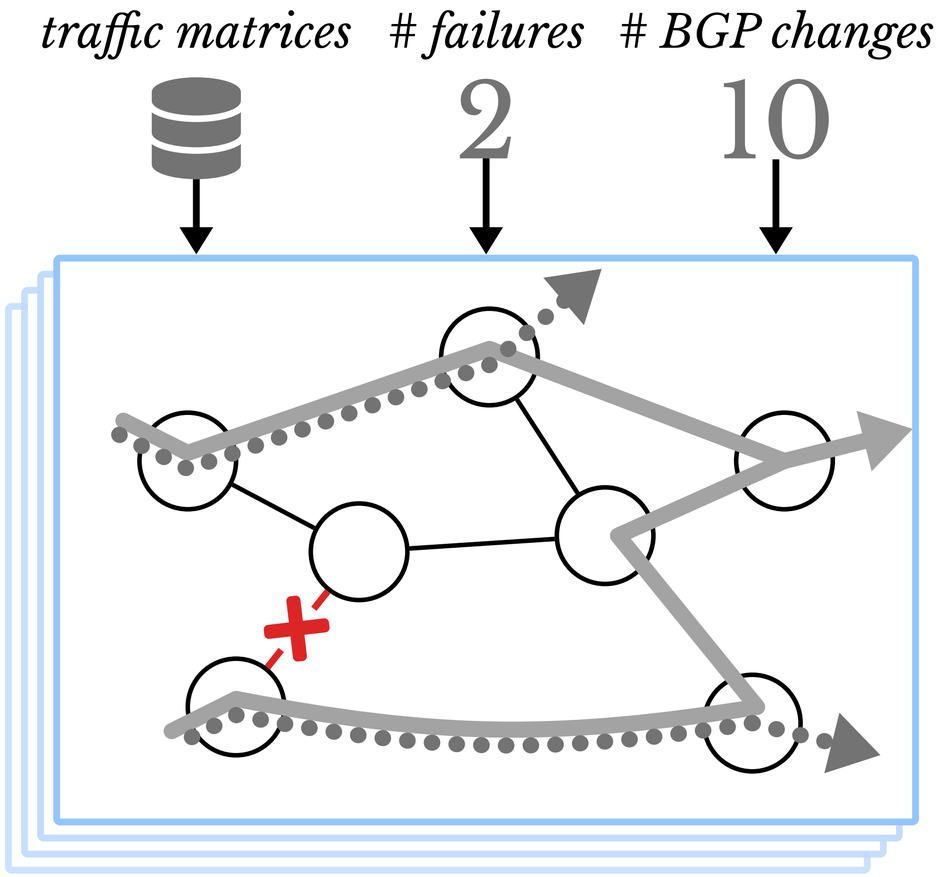
Ignoring routing changes leads to underestimating worst-case loads.



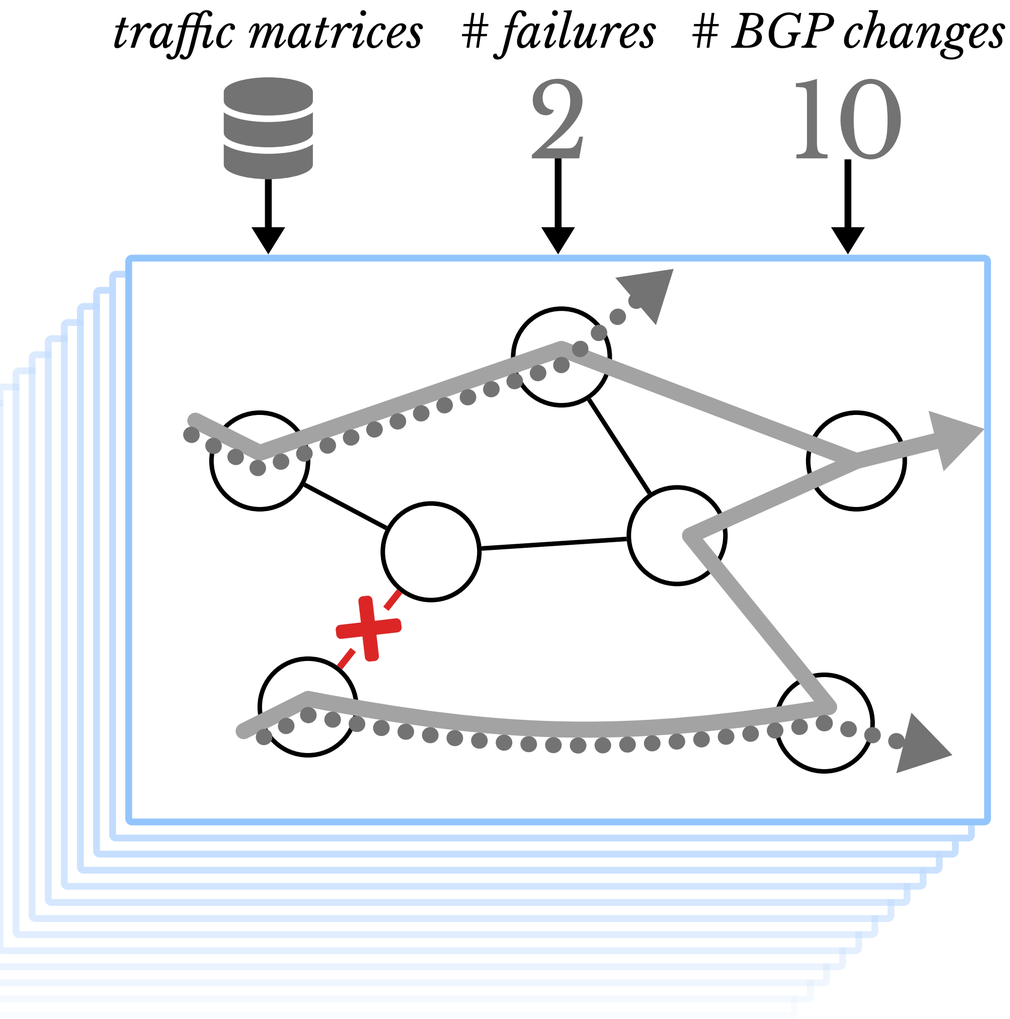
Ignoring routing changes leads to underestimating worst-case loads.



The space of failures *and* route changes is huge and difficult to navigate.

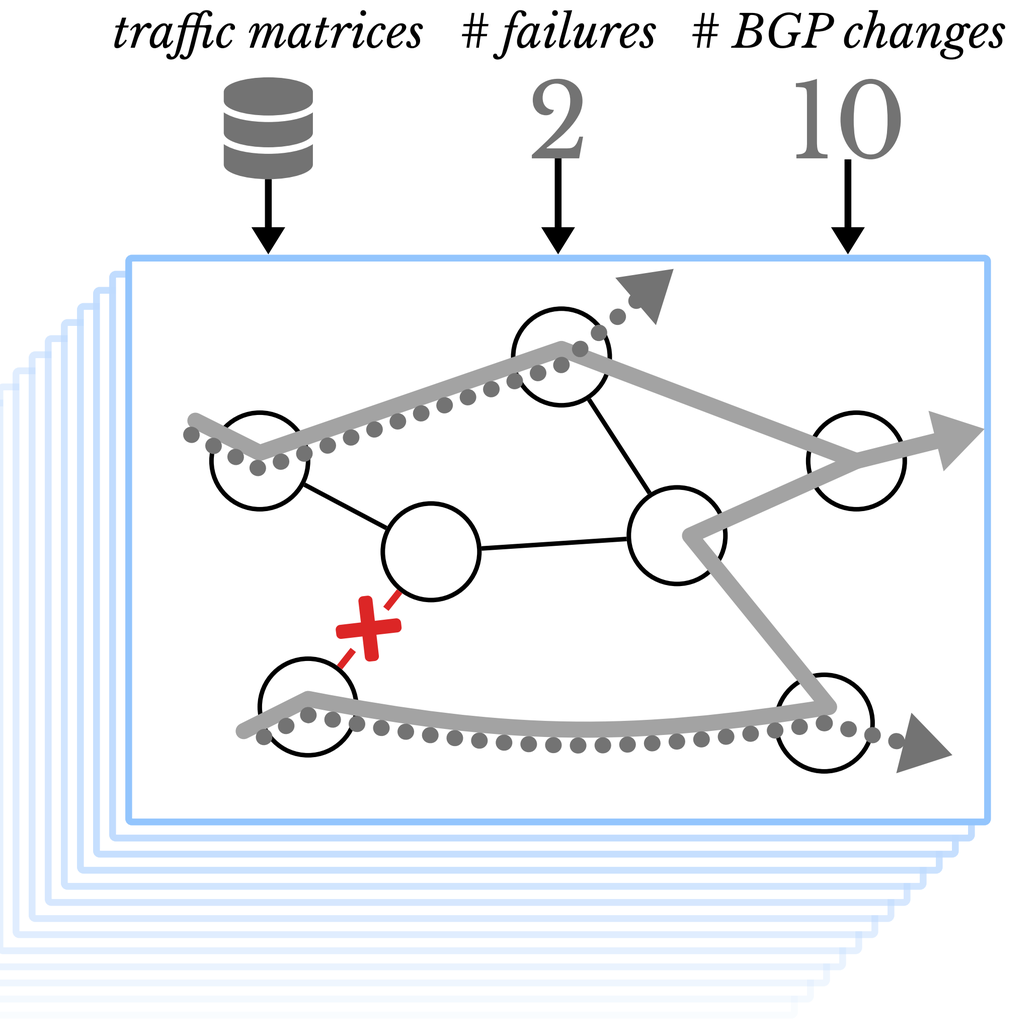


The space of failures *and* route changes is huge and difficult to navigate.



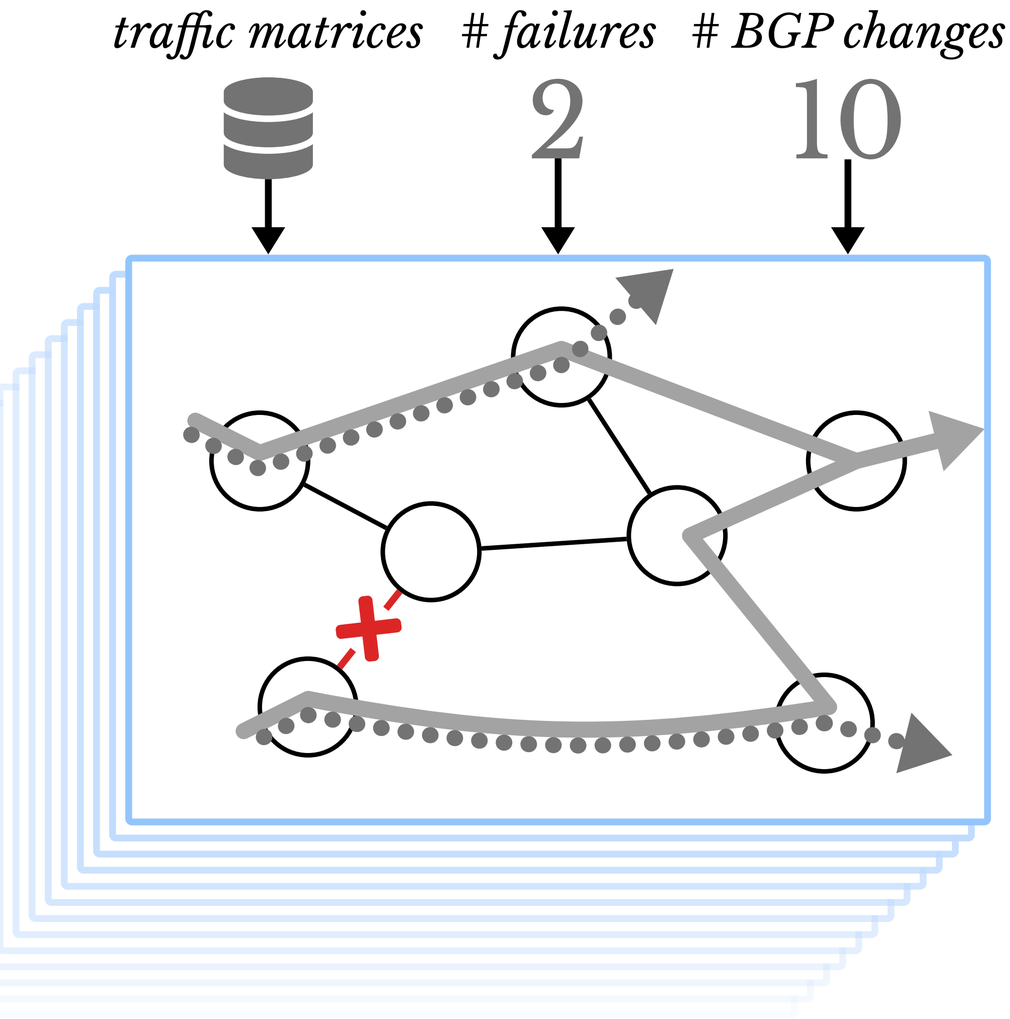
- A destination can be advertised by *any subset* of BGP neighbors.

The space of failures *and* route changes is huge and difficult to navigate.



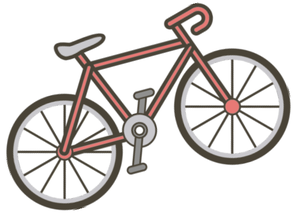
- A destination can be advertised by *any subset* of BGP neighbors.
- Failures create *dependencies* between destinations.

The space of failures *and* route changes is huge and difficult to navigate.



- A destination can be advertised by *any subset* of BGP neighbors.
- Failures create *dependencies* between destinations.
- Over *one million* of destination prefixes.

Velo: **Ve**rify maximum link **lo**ads under failures and routing changes



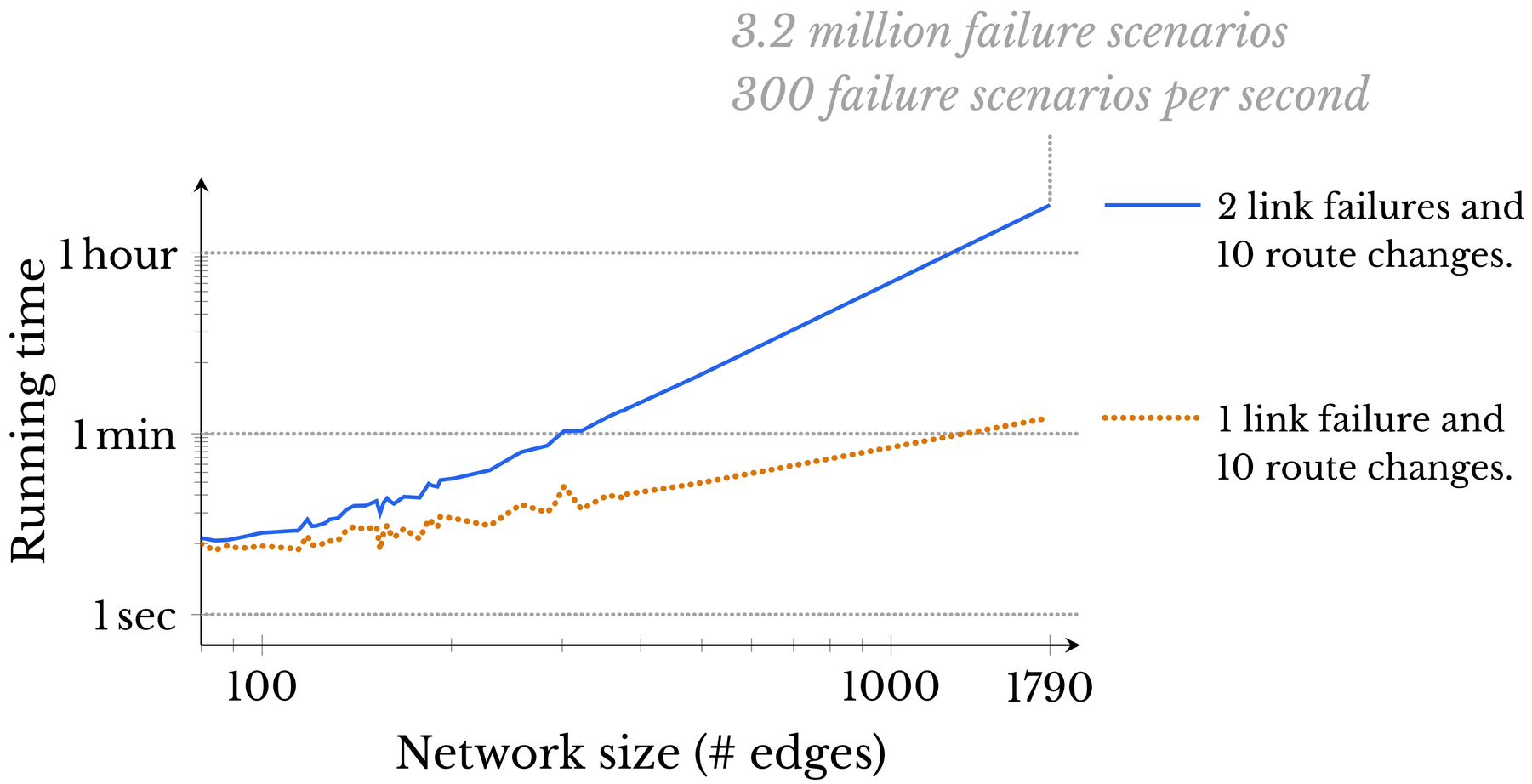
Search space reduction:

A single egress router maximizes link loads.

Input size reduction:

Cluster destination with similar traffic patterns.

Velo finds the worst-case loads in *3 hours* for all 1790 links in an ISP.



SIGCOMM 2020

Probabilistic Verification of Network Configurations

Samuel Steffen
ETH Zurich, Switzerland
samuel.steffen@inf.ethz.ch

Timon Gehr
ETH Zurich, Switzerland
timon.gehr@inf.ethz.ch

Petar Tsankov
ETH Zurich, Switzerland
petar.tsankov@inf.ethz.ch

Laurent Vanbever
ETH Zurich, Switzerland
lvanbever@ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT

Not all important network properties need to be enforced all the time. Often, what matters instead is the fraction of time / probability these properties hold. Computing the probability of a property in a network relying on complex inter-dependent routing protocols is challenging and requires determining all failure scenarios for which the property is violated. Doing so at scale and accurately goes beyond the capabilities of current network analyzers.

In this paper, we introduce NetDice, the first scalable and accurate probabilistic network configuration analyzer supporting BGP, OSPF, ECMP, and static routes. Our key contribution is an inference algorithm to efficiently explore the space of failure scenarios. More specifically, given a network configuration and a property ϕ, our algorithm automatically identifies a set of links whose failure is provably guaranteed not to change whether ϕ holds. By pruning these failure scenarios, NetDice manages to accurately approximate $P(\phi)$. NetDice supports practical properties and expressive failure models including correlated link failures.

We implement NetDice and evaluate it on realistic configurations. NetDice is practical: it can precisely verify probabilistic properties in few minutes, even in large networks.

CCS CONCEPTS

• **Mathematics of computing** → **Probabilistic inference problems**; • **Networks** → **Network properties**.

KEYWORDS

Network analysis, Failures, Probabilistic inference, Cold edges

ACM Reference Format:

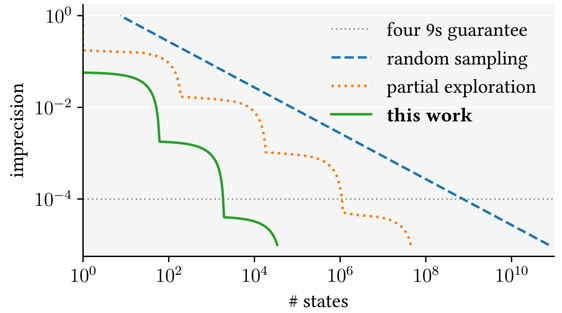


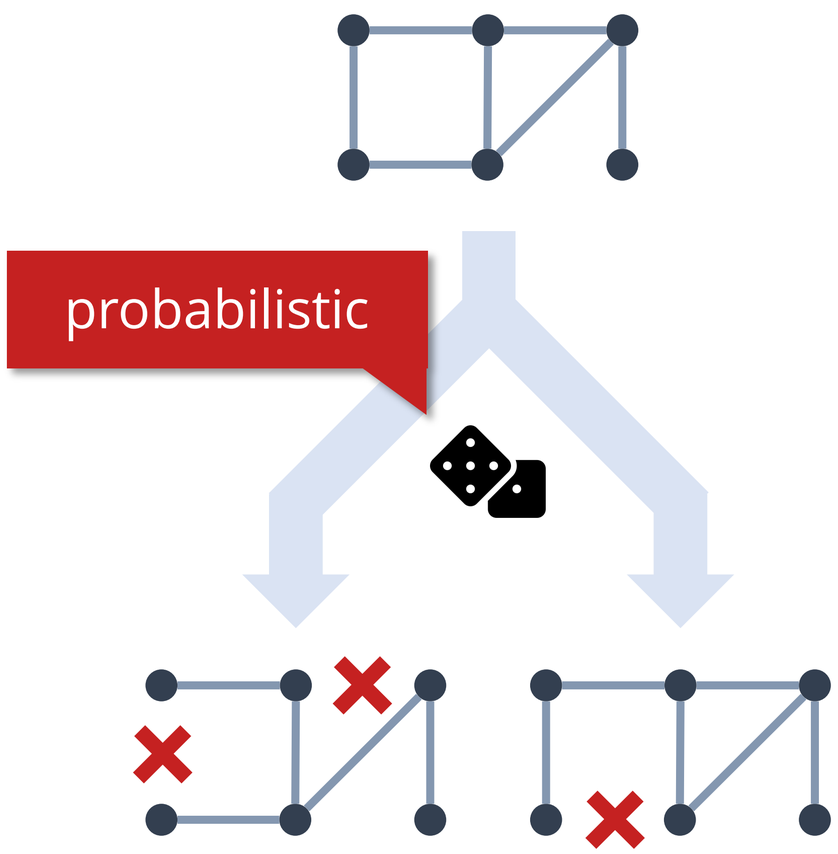
Figure 1: Comparison of approaches for probabilistic network analysis in a network with 191 links and link failure probability 0.001. The confidence for sampling (Hoeffding's inequality) is $\alpha = 0.95$.

1 INTRODUCTION

Ensuring network correctness is an important problem that has received increased attention [1, 4, 12, 16, 19, 31, 40]. So far, existing approaches have focused on verifying "hard" properties, producing a binary answer of whether the property holds under all or a fixed set of failure scenarios.

Besides hard properties, network operators often need to reason about "soft" properties[1] which *can* be violated for a small fraction of time (e.g. 0.01%). Among others, allowing properties to be violated allows for cheaper network designs, e.g. by reducing over-provisioning. Soft properties typically emerge when reasoning about compliance with Service Level Agreements (SLAs). SLAs can be defined with respect to any metric (e.g. path availability, average

Probabilistic Verification



probabilistic

What is the *probability* of ?

high precision required

Service Level Agreements (SLA)
“99.99% reachability”

Traffic Engineering
“80% load-balanced”

“soft” properties

Attempts: Exploring Failures

Too expensive

Partial exploration

1 107 359

#scenarios for *four 9s*,
191 links, $p_{\text{link failure}} = 0.001$

Estimation via sampling

738 M

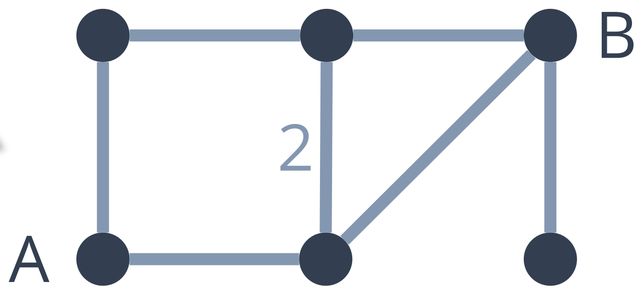
Hoeffding, $\alpha = 0.95$

1 854

≈600x reduction

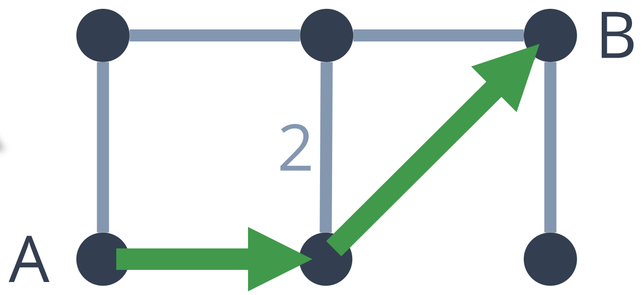
Key Idea

shortest paths



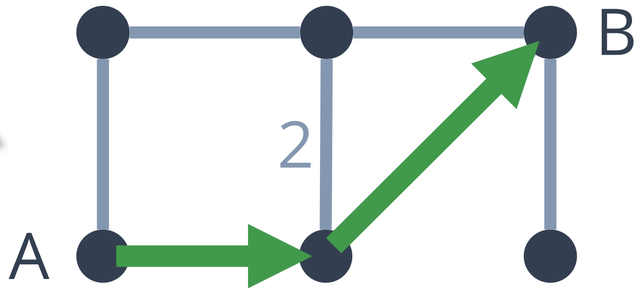
Key Idea

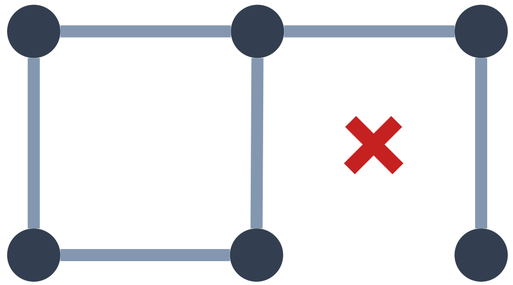
shortest paths



Key Idea

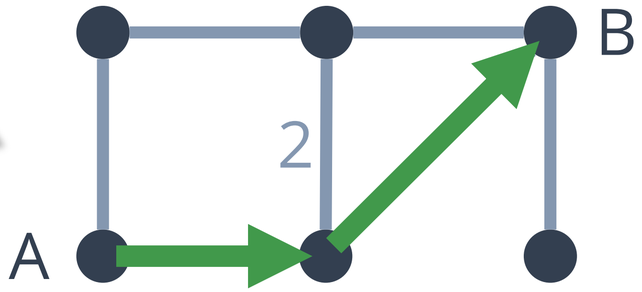
shortest paths

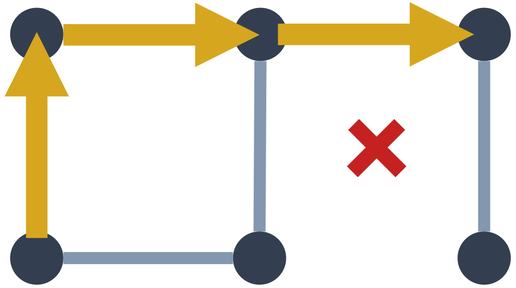




Key Idea

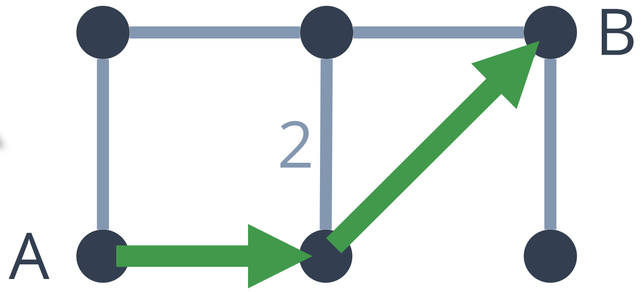
shortest paths

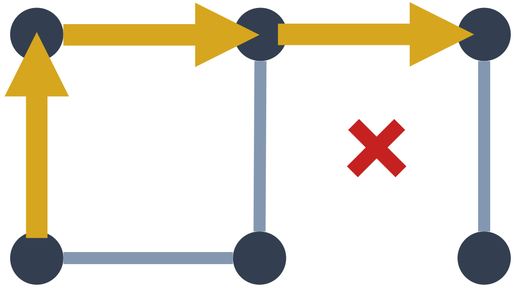


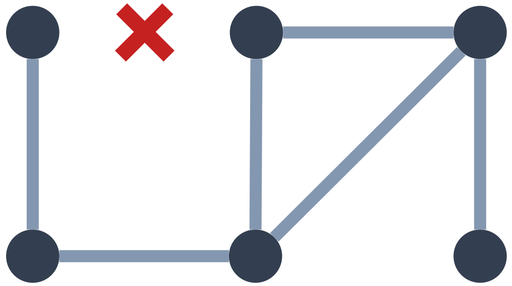


Key Idea

shortest paths

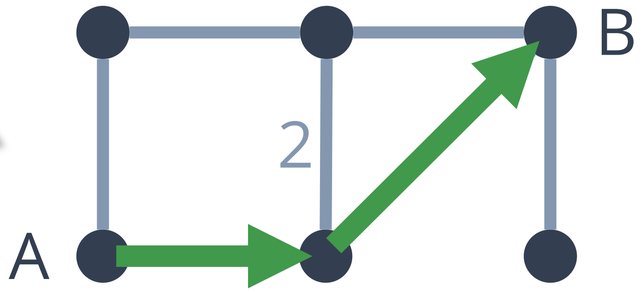


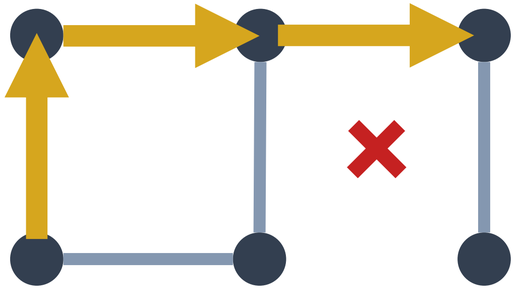


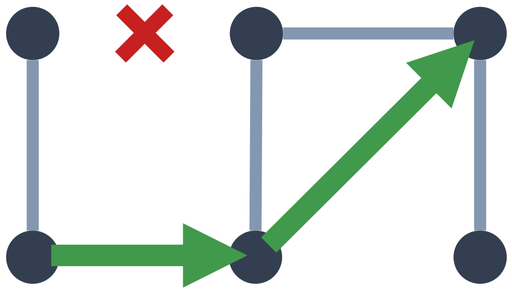


Key Idea

shortest paths

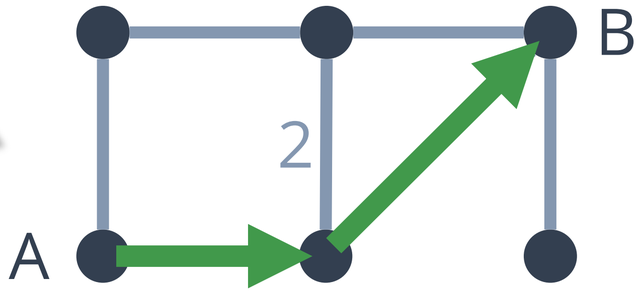


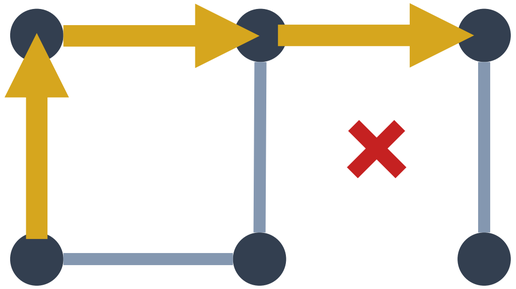




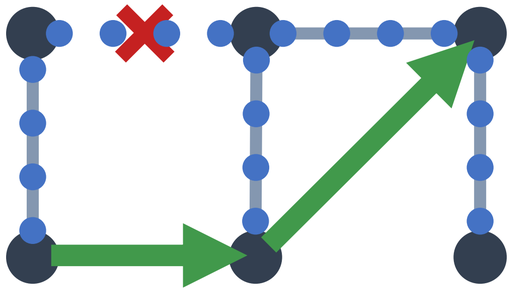
Key Idea

shortest paths



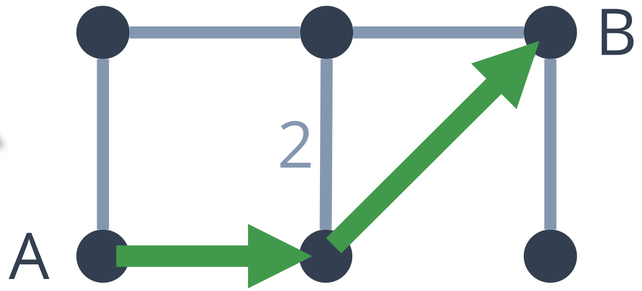


cold edges



Key Idea

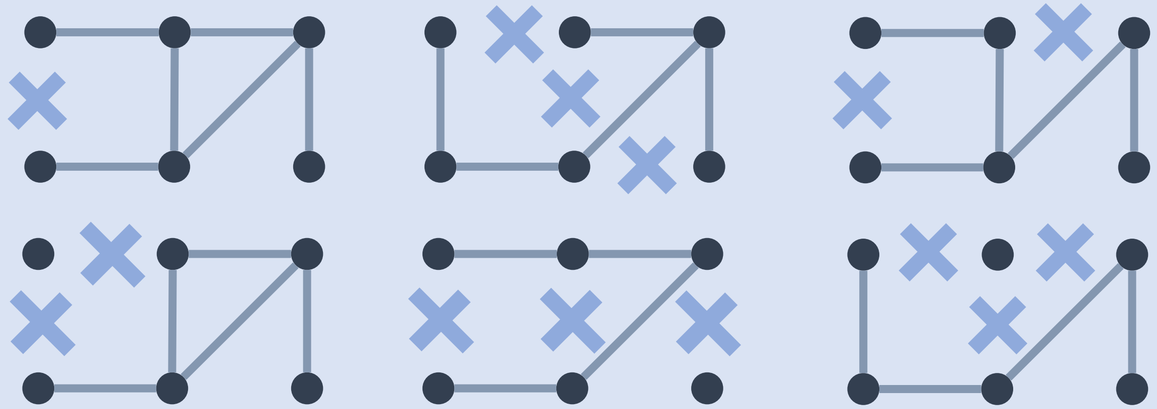
shortest paths



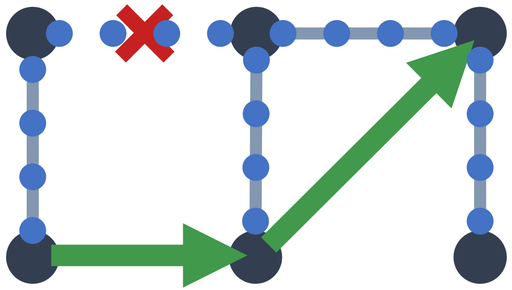
How to find these?

cold edges

Scenarios with same forwarding graph (32 total):



...



Runtime

Single-flow (e.g. Reachability)

Few minutes for *100s* of links for *four 9*s

For 80% of scenarios, > 50% of links are ❄

CoNEXT 2025

Transient Forwarding Anomalies and How to Find Them

ROLAND SCHMID, ETH Zürich, Switzerland
TIBOR SCHNEIDER, ETH Zürich, Switzerland
GEORGIA FRAGKOULI, ETH Zürich, Switzerland
LAURENT VANBEVER, ETH Zürich, Switzerland

Analyzing transient violations of reachability—that happen while routing protocols are re-converging—helps in improving network availability and offering more precise SLAs. The key challenge is analyzing transient violations *accurately*, as they can be short-lived, for *all* affected prefix destinations, and *practically*, without worsening the network's performance. Existing approaches fail to address at least one of these goals: measurement approaches are accurate but only for the prefixes they can probe or observe traffic for, while techniques that estimate the convergence time use the same crude proxy for all prefixes.

To achieve all three goals, we present TRIX, a system that infers transient violation times for BGP events from logged routing events or collected BGP messages. TRIX' key insight is that we do not need to probe all destinations if we use available information to infer the router-local forwarding state, for all destinations, and reconstruct the network-wide violations from router-level state. However, the logged events contain control-plane information that is inaccurate in terms of the content and the times of the forwarding updates, while reconstructing network-wide violations requires reasoning about the flow of traffic through the network. TRIX solves these challenges by simulating the BGP control-plane, modeling the FIB-update rate, and combining the state across routers with propagation delays. To evaluate TRIX, we implement a testbed that relies on a programmable switch and uses 12 real routers. Our evaluation shows that TRIX' inferred reachability violation times are on average within 13–25 ms from the ground truth, and inference scales to large networks.

CCS Concepts: • **Networks** → **Protocol testing and verification**; *Routing protocols*; *Network reliability*; *Formal specifications*.

Additional Key Words and Phrases: Forwarding Anomalies, Routing Convergence, Transient Violations

ACM Reference Format:
Roland Schmid, Tibor Schneider, Georgia Fragkouli, and Laurent Vanbever. 2025. Transient Forwarding Anomalies and How to Find Them. *Proc. ACM Netw.* 3, CoNEXT2, Article 10 (June 2025), 23 pages. https://doi.org/10.1145/3730973

Most network verifiers can only assess correctness properties when the network has converged

Most network verifiers can only assess correctness properties when the network has converged

Corollary

As the network converges, e.g. after a link failure or a route update, important correctness properties might be **violated**

Most network verifiers can only assess correctness properties when the network has converged

Corollary

As the network converges, e.g. after a link failure or a route update, important correctness properties might be **violated**

Research question

How long are these transient violations? How often do they appear?

Accurately estimating these violation times is hard

Routing events can affect any prefix(es)

many

Violations can be short-lived, or not

from milliseconds to many seconds

Accurately estimating these violation times is hard

Routing events can affect any prefix(es)

many

Violations can be short-lived, or not

from milliseconds to many seconds

Active probing is not realistic

Our system infers violation times accurately and for all prefixes, from widely available control-plane logs

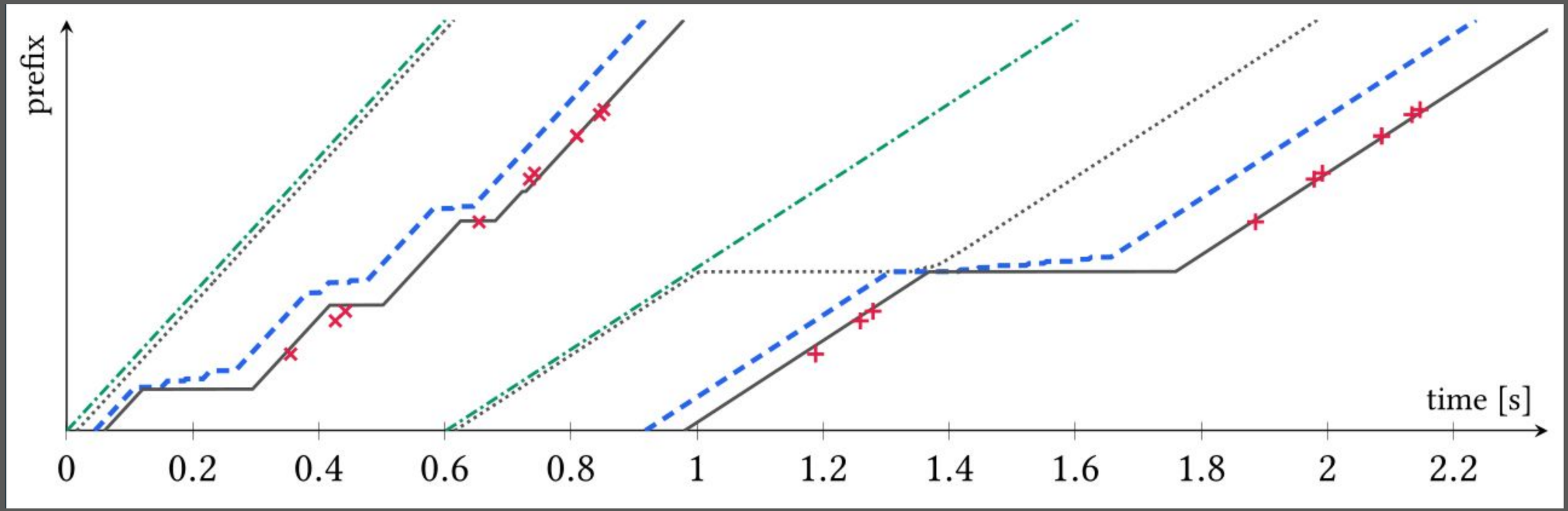
Our system infers violation times accurately and for all prefixes,

from widely available control-plane logs

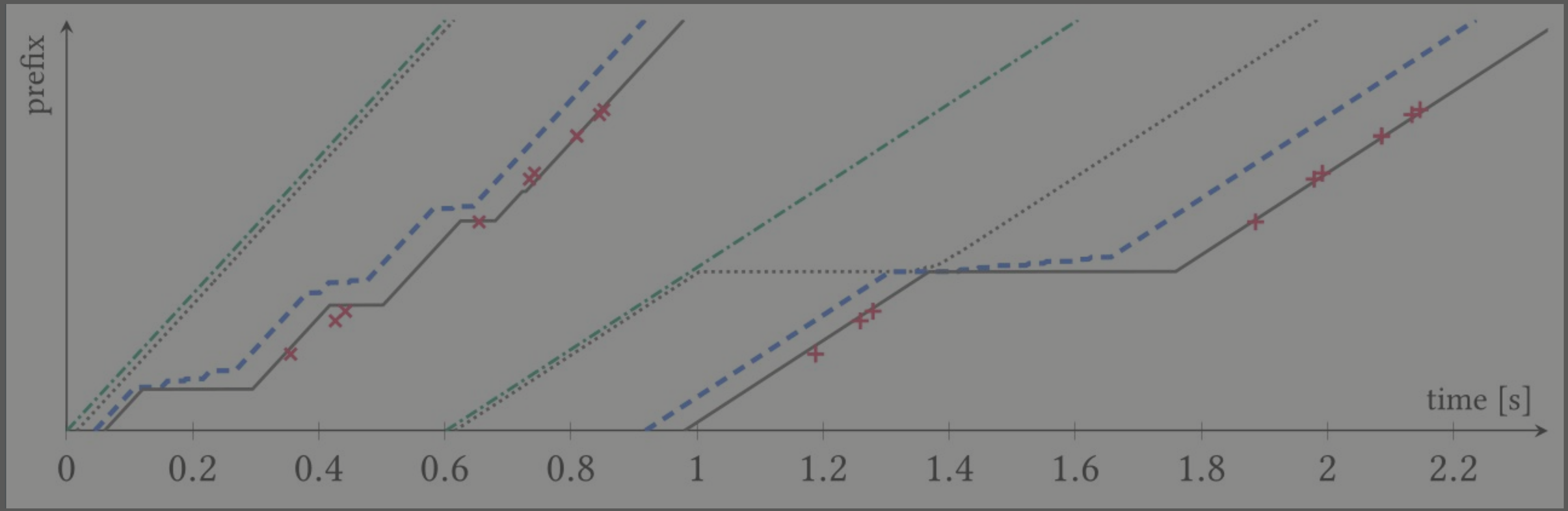
BGP events, RIB events, FIB writes, …

The key challenge is to bridge the gap between
control plane events and their effects in the data plane (if any)

Our system bridges this gap by modeling the RIB-to-FIB bottleneck

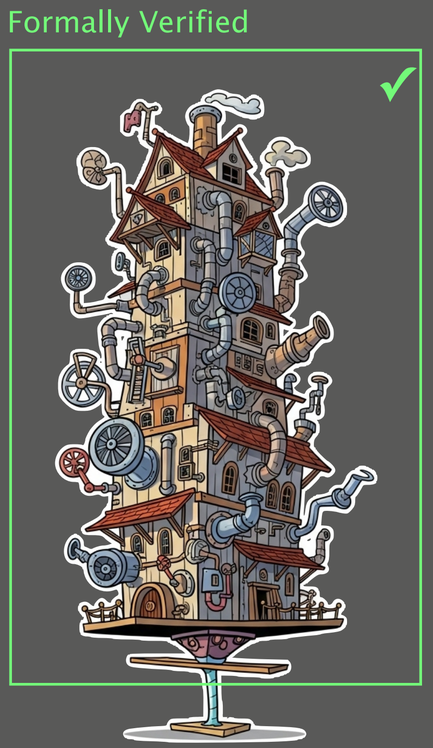


Our system achieves an average accuracy of 13–25ms



Verifying configurations was the easy part

My ongoing quest toward correct network operations



More properties

beyond reachability

More coverage

beyond handcrafted models

Better usability

beyond human intervention

A network verifier can only be blindly trusted if its analysis is **sound** and **complete**

A network verifier can only be blindly trusted if its analysis is **sound** and **complete**

sound	Reports **all** bugs
complete	Reports **only** bugs

Ensuring both is often impossible (Rice's theorem)

verifiers need to accurately model…

all protocols and *all* their behaviors
BGP, OSPF, IS-IS, EIGRP, …

for all vendors, devices and OSes
Cisco, Juniper, Brocade, Arista, …

Ensuring both is often impossible (Rice's theorem)

verifiers need to
accurately model…

all protocols and *all* their behaviors
BGP, OSPF, IS-IS, EIGRP, …

for all vendors, devices and OSes
Cisco, Juniper, Brocade, Arista, …

… including any bugs

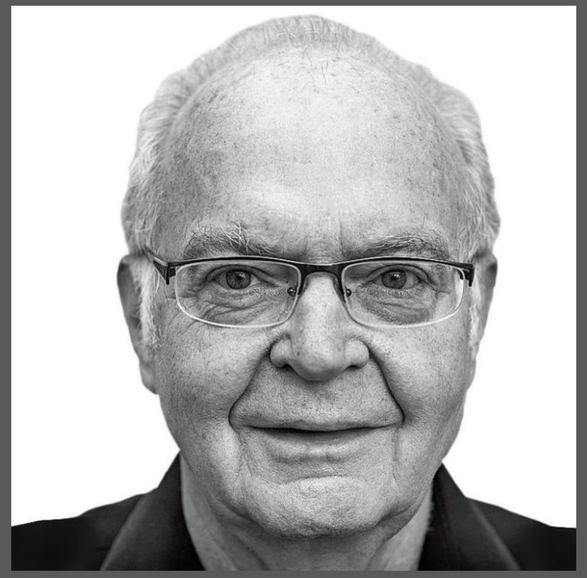


Photo: IEEE Computer Society

"Beware of bugs in the above code;
I have only proved it correct,
not tried it."

—Donald Knuth

detect model
inaccuracies

build
accurate models

automatically

detect model
inaccuracies

build
accurate models

automatically

NSDI 2021

Metha: Network Verifiers Need To Be Correct Too!

Rüdiger Birkner* Tobias Brodmann* Petar Tsankov Laurent Vanbever Martin Vechev

*These authors contributed equally to this work.

ETH Zürich

Abstract

Network analysis and verification tools are often a godsend for network operators as they free them from the fear of introducing outages or security breaches. As with any complex software though, these tools can (and often do) have bugs. For the operators, these bugs are not necessarily problematic except if they affect the precision of the model as it applies to their specific network. In that case, the tool output might be wrong: it might fail to detect actual configuration errors and/or report non-existing ones.

In this paper, we present Metha, a framework that systematically tests network analysis and verification tools for bugs in their network models. Metha automatically generates syntactically- and semantically-valid configurations; compares the tool's output to that of the actual router software; and detects any discrepancy as a bug in the tool's model. The challenge in testing network analyzers this way is that a bug may occur very rarely and only when a specific set of configuration statements is present. We address this challenge by leveraging grammar-based fuzzing together with combinatorial testing to ensure thorough coverage of the search space and by identifying the minimal set of statements triggering the bug through delta debugging.

We implemented Metha and used it to test three well-known

This fictitious situation illustrates an intrinsic problem with validation technologies: their results can only be completely trusted if their analysis is sound and complete. As with any complex software though, these tools can (and often do) have bugs. To be fair, 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, every device can exhibit slightly different behaviors under certain conditions. For all it takes, these behaviors might be the results of bugs themselves. And yet, failing to accurately capture these behaviors—as we show—can lead to incorrect and possibly misleading analysis results.

A fundamental and practical research question is therefore: *How can developers make sure that their network analysis and verification tools are correct?*

Metha We introduce Metha, a system that thoroughly tests network analysis and verification tools to find subtle bugs in their network models using black-box differential testing. Metha automatically finds model discrepancies by generating input configurations and comparing the output of the tool under test with the output produced by the actual router software. For every discovered discrepancy, Metha provides a minimal

Metha systematically tests network verifiers through automated configuration generation

Metha

Supply configurations
and compare results

Tool under test

Oracle

Metha: Automated Testing of Network Analyzers

1. **Sensible configurations**
   satisfying configuration dependencies
2. **Systematic exploration**
   covering the search space thoroughly
3. **Actionable feedback**
   isolating the problematic statements

Metha found bugs in all three tested tools

	# bugs	crash	silent
Batfish	30	5	25
NV	30	5	25
C-BGP	3	0	3

detect model
inaccuracies

build
accurate models

automatically

NeurIPS 2022

Learning to Configure Computer Networks with Neural Algorithmic Reasoning

Luca Beurer-Kellner[1, *] **Martin Vechev**[1] **Laurent Vanbever**[1] **Petar Veličković**[2]

[1]ETH Zurich, Switzerland [2]DeepMind

https://github.com/eth-sri/learning-to-configure-networks

Abstract

We present a new method for scaling automatic configuration of computer networks. The key idea is to relax the computationally hard search problem of finding a configuration that satisfies a given specification into an approximate objective amenable to learning-based techniques. Based on this idea, we train a neural algorithmic model which learns to generate configurations likely to (fully or partially) satisfy a given specification under existing routing protocols. By relaxing the rigid satisfaction guarantees, our approach (i) enables greater flexibility: it is protocol-agnostic, enables cross-protocol reasoning, and does not depend on hardcoded rules; and (ii) finds configurations for much larger computer networks than previously possible. Our learned synthesizer is up to 490× faster than state-of-the-art SMT-based methods, while producing configurations which on average satisfy more than 92% of the provided requirements.

1 Introduction

Configuring large-scale networks is a challenging and important task as network configuration mistakes regularly lead to massive internet-wide outages affecting millions (resp. billions[2]) of Internet users [35, 25]. Typically, network operators provide a router-level configuration W which, after applying protocols such as shortest-path routing, induces a certain forwarding behaviour FWD as illustrated in Figure 1. As this remains a challenging task, much recent research has focused on automating configuration by leveraging synthesis techniques [15, 5, 31]: A synthesizer is used to

Can we learn how to configure a network?

Can we learn how to configure a network?

Yes, to an extent

Train a graph-based neural model
to invert network computations

Using generated pairs of (cfgs, specs)
simulate the cfg, extract the forwarding state

Fully automatic!

Neural-based configuration synthesis is much faster but does not yet achieve 100% accuracy

Result BGP/OSPF configuration synthesis

- **Learning-based synthesis** is much faster than SOTA SMT tools **(490x)** while still maintaining good consistency (**92% of req.**)
- Other benefits: **UNSAT specification, sample variations**

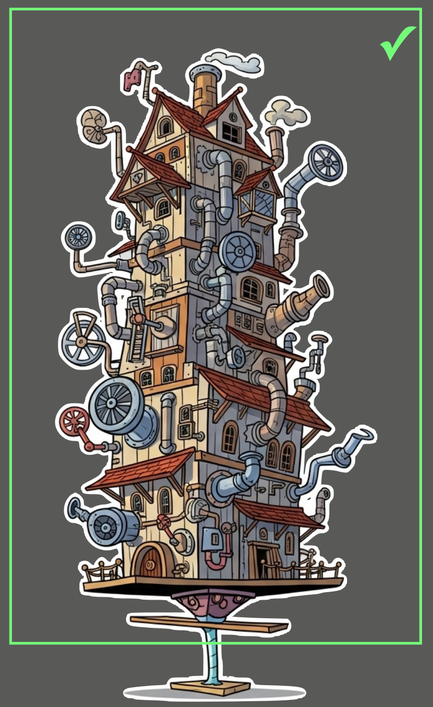
Table 3: Comparing consistency and synthesis time of our method (Neural) with the SMT-based NetComplete. The notation n/8 TO indicates the number of timed out runs out of 8 (25+ minutes).

# Requirements		NetComplete (s)	Neural CPU (s)	Speedup	∅ Consistency	Full Matches
2 reqs.	S	18.07s±14.55	0.72s±0.54	**25.2x**	0.97±0.09	7/8
	M	60.86s±33.39	3.18s±4.32	**19.1x**	0.94±0.13	6/8
	L	1389.48s±312.58 7/8 TO	24.25s±28.35	**57.3x**	0.99±0.03	7/8
8 reqs.	S	247.69s±436.90	1.25s±1.02	**198.7x**	0.96±0.08	6/8
	M	>25m 8/8 TO	4.55s±4.30	**329.8x**	0.97±0.04	4/8
	L	>25m 8/8 TO	31.28s±28.53	**48.0x**	0.97±0.05	5/8
16 reqs.	S	1416.83s±235.25 7/8 TO	2.88s±1.66	**492.0x**	0.92±0.06	1/8
	M	>25m 8/8 TO	6.53s±5.10	**229.8x**	0.95±0.05	2/8
	L	>25m 8/8 TO	87.99s±141.97	**17.0x**	0.95±0.03	2/8

Verifying configurations was the easy part

My ongoing quest toward correct network operations

Formally Verified



More properties

beyond reachability

More coverage

beyond handcrafted models

Better usability

beyond human intervention

To go mainstream, network verification needs to be more usable

even assuming more properties and coverage

Clarify *what* to verify

Simplify the "before"

Deploy verified configurations

Simplify the "after"

Clarify *what*
to verify

Deploy verified
configurations

Simplify the "before"

NSDI 2020

Config2Spec: Mining Network Specifications from Network Configurations

Rüdiger Birkner[1] Dana Drachsler-Cohen[2*] Laurent Vanbever[1] Martin Vechev[1]

[1]*ETH Zürich* [2]*Technion*

Abstract

Network verification and configuration synthesis are promising approaches to make networks more reliable and secure by enforcing a set of policies. However, these approaches require a formal and precise description of the intended network behavior, imposing a major barrier to their adoption: network operators are not only reluctant to write formal specifications, but often do not even know what these specifications are.

We present *Config2Spec*, a system that automatically synthesizes a formal specification (a set of policies) of a network given its configuration and a failure model (e.g., up to two link failures). A key technical challenge is to design a synthesis algorithm which can efficiently explore the large space of possible policies. To address this challenge, *Config2Spec* relies on a careful combination of two well-known methods: data plane analysis and control plane verification.

Experimental results show that *Config2Spec* scales to mining specifications of large networks (>150 routers).

1 Introduction

Consider the task of a network operator who—tired of human-induced network downtimes—decides to rely on formal meth-

homegrown over years, by a team of network engineers (some of which do not even work there anymore).

This situation illustrates the difficulty of writing network specifications. Akin to software specifications, formal specifications are hard to write (as hard as writing the program in the first place [20]), debug, and modify [2, 21]. Yet, without easier ways to provide network specifications, network verification and synthesis are unlikely to get widely deployed.

Config2Spec We introduce *Config2Spec*, a system that automatically mines a network's specification from its configurations and a failure model (e.g., up to k failures). *Config2Spec* is precise: it returns *all* policies that hold under the failure model (no false negatives) and *only* those (no false positives).

Challenges Mining precise network specifications is challenging as it involves exploring two exponential search spaces: *(i)* the space of all possible policies, and *(ii)* the space of all possible network-wide forwarding states. The challenge stems from the fact that individually exploring each of the search spaces can be prohibitive: a search for the true policies is hard since they are a small fraction of the policy space, while a search for the violated policies is hard since these require witnesses (data planes), which are often sparse.

Insights *Config2Spec* addresses the above challenges by com-

Internet2's specification with its 10 routers consists of ~4000 policy predicates.

Config2Spec mines the network's full specification from its configuration and the required failure tolerance

Clarify *what* to verify

Deploy verified configurations

Simplify the "after"

SIGCOMM 2023

Taming the transient while reconfiguring BGP

Tibor Schneider
ETH Zurich
sctibor@ethz.ch

Roland Schmid
ETH Zurich
roschmi@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@ucl.ac.uk

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

ABSTRACT

BGP reconfigurations are a daily occurrence for most network operators, especially in large networks. Yet, performing safe and robust BGP reconfiguration changes is still an open problem. Few BGP reconfiguration techniques exist, and they are either *(i) unsafe*, because they ignore transient states, which can easily lead to invariant violations; or *(ii) impractical*, as they duplicate the entire routing and forwarding states, and require special hardware.

In this paper, we introduce Chameleon, the first BGP reconfiguration framework capable of maintaining correctness throughout a reconfiguration campaign while relying on standard BGP functionalities and minimizing state duplication. Akin to concurrency coordination in distributed systems, Chameleon models the reconfiguration process with happens-before relations. This modeling allows us to capture the safety properties of transient BGP states. We then use this knowledge to precisely control the BGP route propagation and convergence, so that input invariants are provably preserved at any time during the reconfiguration.

We fully implement Chameleon and evaluate it in both testbeds and simulations, on real-world topologies and large-scale reconfiguration scenarios. In most experiments, our system computes reconfiguration plans within a minute, and performs them from start to finish in a few minutes, with minimal overhead.

CCS CONCEPTS

• **Networks** → **Network management**; **Routing protocols**; *Network control algorithms*; Network reliability.

KEYWORDS

Border Gateway Protocol (BGP), reconfiguration, network update, convergence, scheduling

ACM Reference Format:

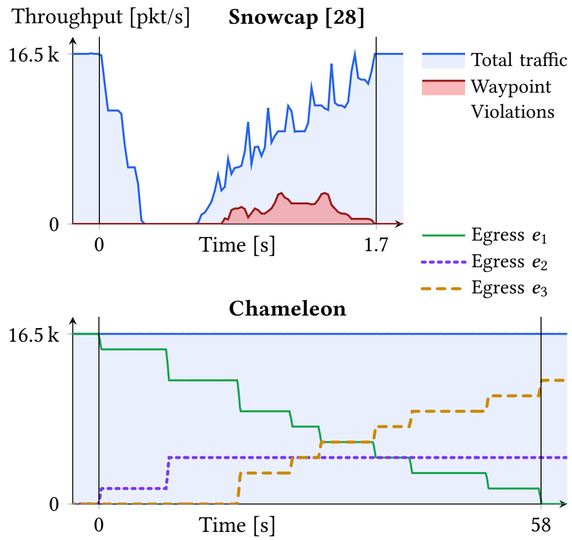


Figure 1: BGP reconfigurations often lead to disruptions, even when using recent reconfiguration frameworks such as Snowcap [28]. Here, Snowcap transiently violates two invariants (reachability and waypointing). In contrast, Chameleon reconfigures the network without violating any.

1 INTRODUCTION

Much has been written about network reconfigurations, their frequency [12, 20, 28, 32, 36] and their disruptiveness [18, 22, 28]. Yet, reconfiguration-induced downtimes *still* happen. In fact, Alibaba recently stated that the *majority* of their network outages resulted from configuration updates [22].

Among all reconfiguration scenarios, BGP ones are special be-

Network reconfiguration

Network reconfiguration

Given specification φ

Network reconfiguration

Given specification φ and an initial configuration C_i

final C_f

Network reconfiguration

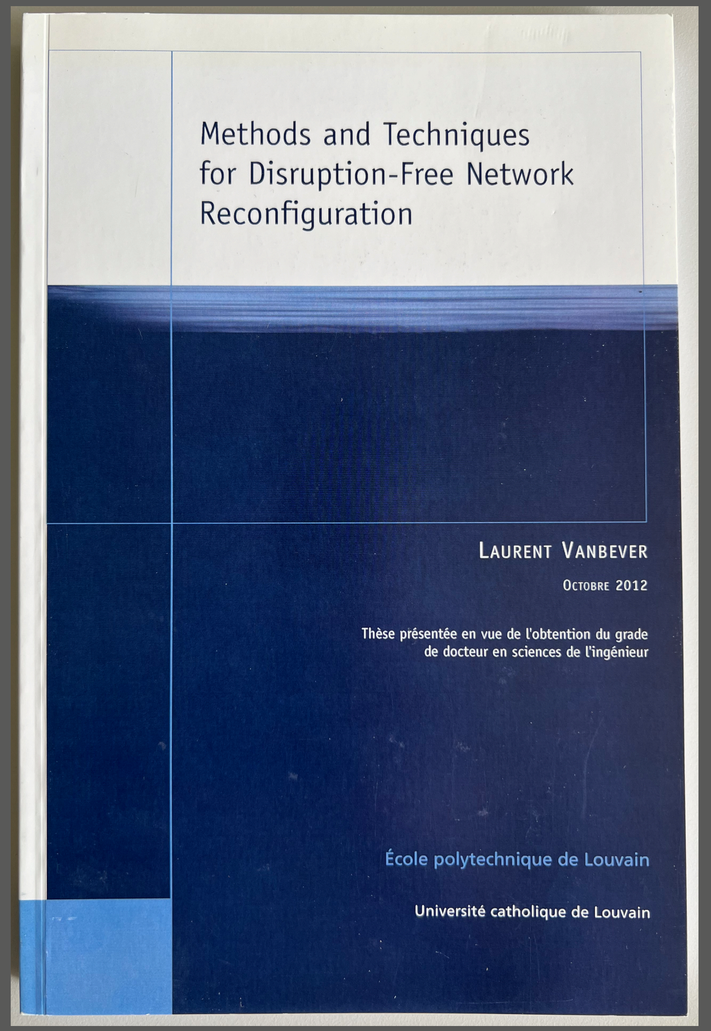
Given specification φ and an initial configuration C_i

final C_f

Return $C_i, C_a, C_b, \ldots C_f \models \varphi$

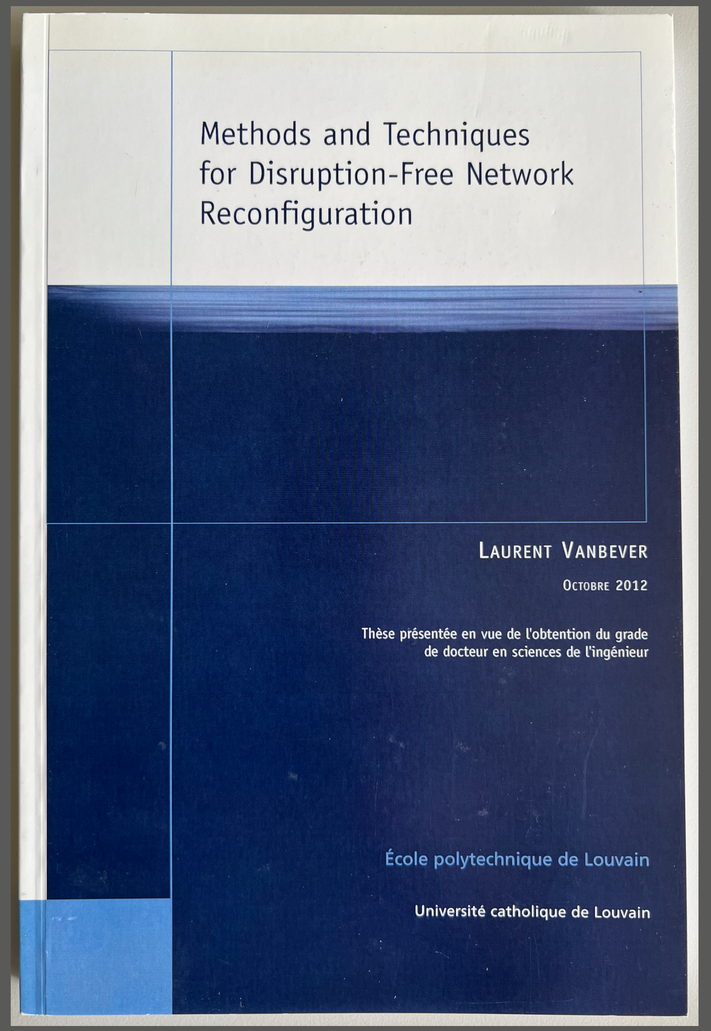
Not exactly a new problem…

Not exactly a new problem…

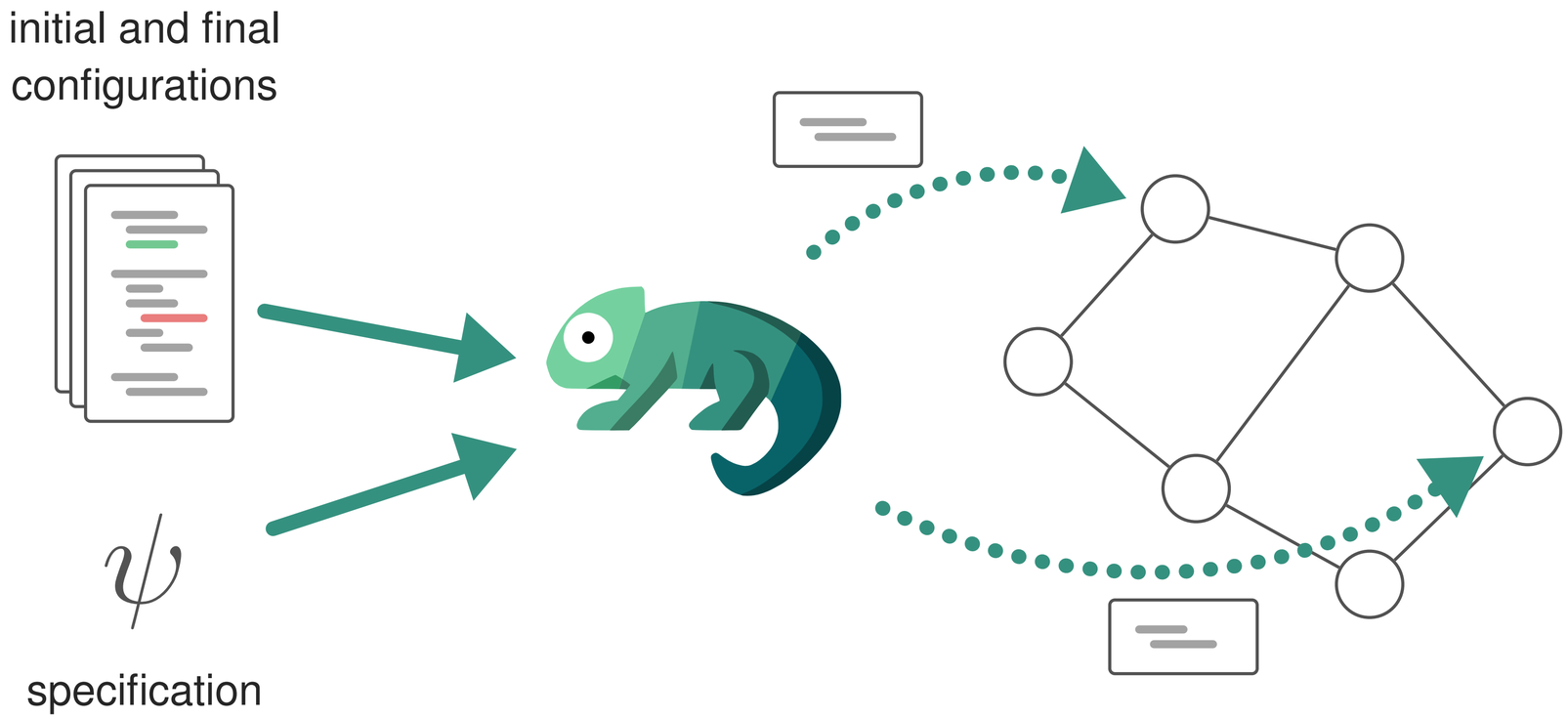


Existing techniques...

- duplicate the entire control plane;
- temporarily "freeze" it" (cannot react to failures)



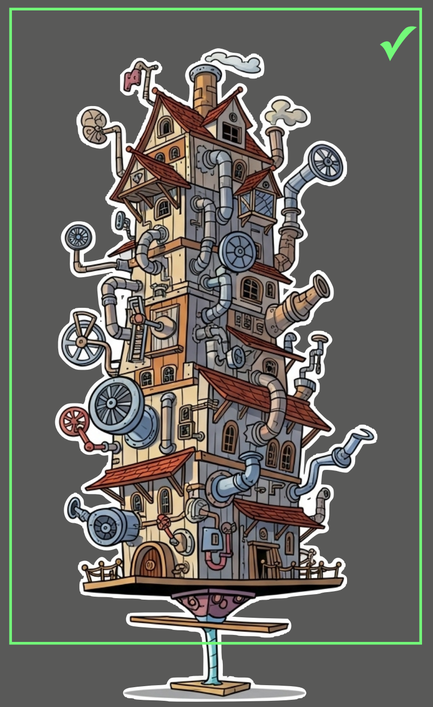
Chameleon reconfigures networks in-place, one router at a time, correctly, and within 5 minutes even for large networks



Verifying configurations was the easy part

My ongoing quest toward correct network operations

Formally Verified



More properties

beyond reachability

More coverage

beyond handcrafted models

Better usability

beyond human intervention

Configuring a network is only a small part of network operations as a whole

Configuring a network is only a small part of network operations as a whole

Network operations

Configure

Design

Monitor

Optimize

Secure

Troubleshoot

and *many* others…

Similarly to configuring,

many of these tasks are bottlenecked by human reasoning

Configure

Design

Monitor

Optimize

Secure

Troubleshoot

Similarly to configuring, one can approach these problems in a model-based or a model-less way

Similarly to configuring, one can approach these problems in a model-based or a model-less way

model-based

model-less

"AI"

Similarly to configuring, one can approach these problems in a model-based or a model-less way

	model-based	model-less "AI"
Guarantees		
Coverage		
Explainability		
Overhead		

Similarly to configuring, one can approach these problems in a model-based or a model-less way

	model-based	model-less "AI"
Guarantees	strong	
Coverage	reduced	
Explainability	high	
Overhead	high upstream	

Similarly to configuring, one can approach these problems in a model-based or a model-less way

	model-based	model-less "AI"
Guarantees	strong	few
Coverage	reduced	high
Explainability	high	poor
Overhead	high upstream	high downstream

Similarly to configuring, one can approach these problems in a model-based or a model-less way

	model-based	model-less "AI"
Guarantees	strong	few
Coverage	reduced	high
Explainability	high	poor
Overhead	high upstream	high downstream

Can we get the best of both worlds?

model-based

X

model-less

SIGCOMM 2025

POSTER: Continual Benchmarking of LLM-Based Systems on Networking Operations

Ioannis Protogeros
ETH Zürich
iprotogeros@ethz.ch

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

Abstract

The inherent complexity of operating modern network infrastructures has led to growing interest in using Large Language Models (LLMs) to support network operators, particularly in the area of Incident Management (IM). Yet, the absence of standardized benchmarks for evaluating such systems poses challenges in tracking progress, comparing approaches, and uncovering their limitations. As LLM-based tools become widespread, there is a clear need for a comprehensive benchmarking suite that reflects the diversity and complexity of operational tasks encountered in real-world networks.

This poster outlines our vision for designing such a modular benchmarking suite. We describe an approach for generating operational tasks of varying complexity and discuss how to evaluate LLMs on these tasks and assess system-level performance. As a preliminary evaluation, we benchmark three LLMs — GPT-4.1, Gemini 2.5-Pro, and Claude 3.7 Sonnet — across over 100 test cases and two pipeline variants.

CCS Concepts

• **Networks** → **Network management**; • **Computing methodologies** → **Knowledge representation and reasoning**; **Artificial intelligence**.

Keywords

Large Language Models, Network Management, Benchmarking, Incident Management

ACM Reference Format:
Ioannis Protogeros and Laurent Vanbever. 2025. POSTER: Continual Benchmarking of LLM-Based Systems on Networking Operations. In *ACM SIGCOMM 2025 Conference (SIGCOMM Posters and Demos '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3744969.3748425

well-defined tasks of **varying complexity** remains a challenging task in the inherently complex setting of Network IM.

Focusing on the case of misconfigured networks disrupting their intended behaviour, we pose the following Research Questions (RQs) that capture the requirements of a benchmark in our task context:

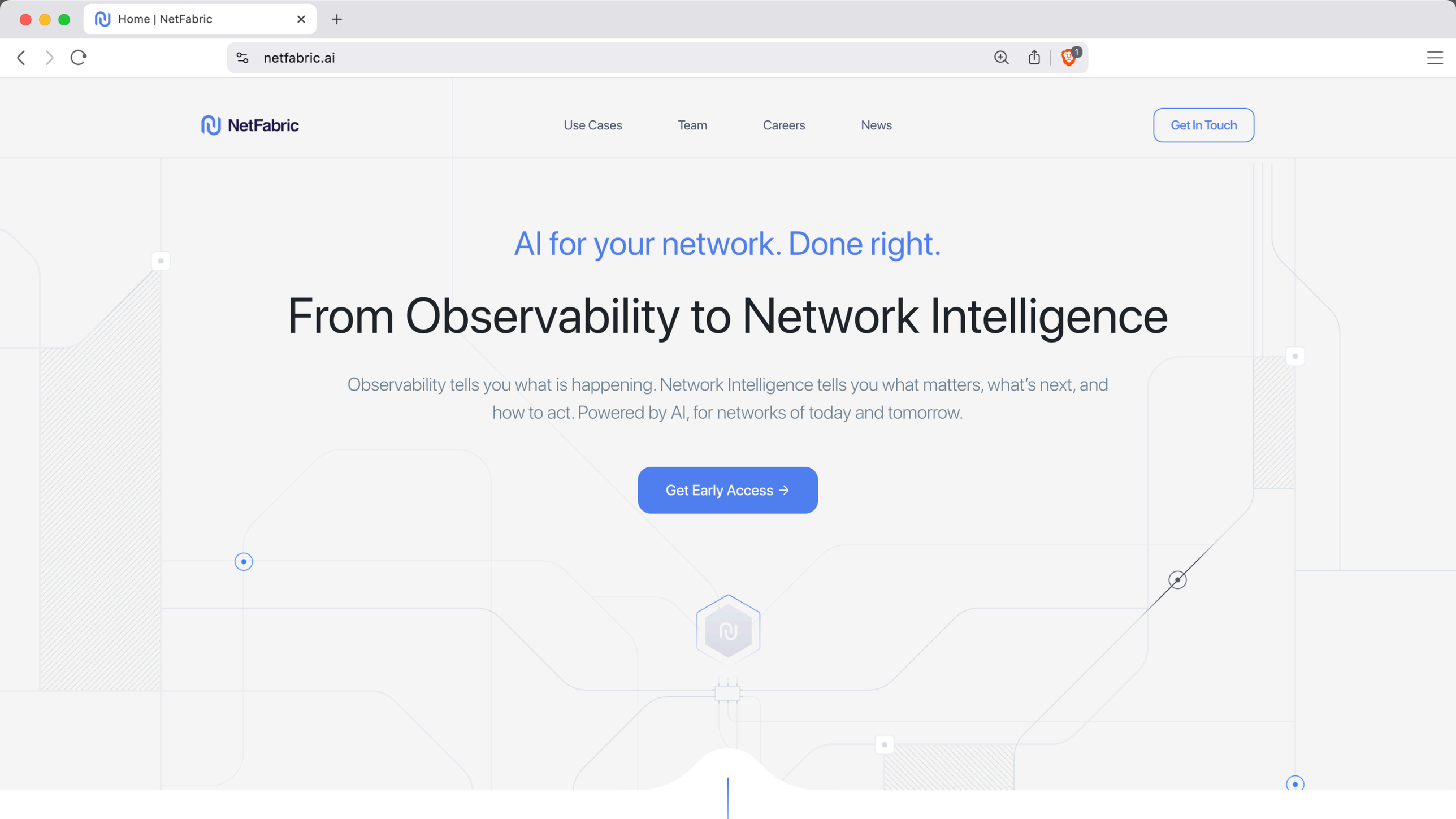
RQ1: *How can we automatically generate realistic and complex fault scenarios and quantify their impact on network behaviour?* Modeling disruption severity and resolution difficulty from misconfigurations requires establishing nonlinear causal relationships. Capturing this complexity is paramount for generating challenging and meaningful test cases.

RQ2: *(i) How should network state be encoded for LLM-based systems, and (ii) which solution pipelines are most effective?* Various approaches are being proposed, from the handling of raw network data [4, 6] to the definition of structured workflows integrating specialized tools [3]. These strategies require evaluation under a unified and extensible testbed to ensure comparability and relevance.

RQ3: *How do we meaningfully assess an LLM's proposed fix against a "ground-truth", intended network behavior?* Using data to represent the network states, we need to distill metrics that succinctly describe a remedy's efficacy.

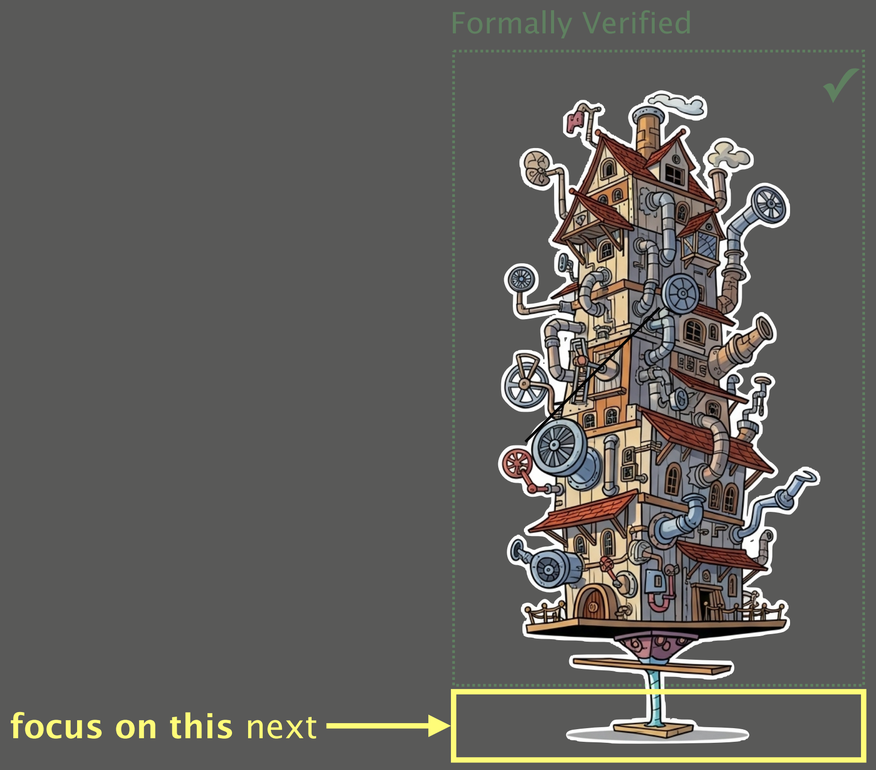
This work identifies key design considerations for automated LLM-centered system evaluation in Network Incident Management. We present a system that allows the *continual* incorporation of the advancements related to these open research questions. We also highlight the importance of extensibility when evaluating complex workflows that cannot be effectively addressed with a monolithic *Input* → *LLM* → *Output* design.

2 Proposed Framework and Design Considerations


Home | NetFabric
netfabric.ai
NetFabric
Use Cases
Team
Careers
News
Get In Touch
AI for your network. Done right.
From Observability to Network Intelligence
Observability tells you what is happening. Network Intelligence tells you what matters, what's next, and how to act. Powered by AI, for networks of today and tomorrow.
Get Early Access →

Verifying configurations was the easy part

My ongoing quest toward correct network operations



Laurent Vanbever

nsg.ethz.ch | netfabric.ai

FMANO

Mon Sept 8 2025