Towards Accessible Model-Free Verification

Alexander Krentsel*
Google / UC Berkeley

Oliver Ye
Google

Anthony Tafoya
Google

Xuqian Ma
Google

Sylvia Ratnasamy
Google / UC Berkeley

Anees Shaikh
Google

Abstract

Despite coming up on two decades of network verification research, verification tooling continues to see limited real-world adoption and outages continue to occur. Relying on interviews with network engineers and our own experience as a large network operator, we ask *why*. These conversations reveal that the culprit is traditional verification's reliance on hand-crafted network models, which leads to issues with coverage, correctness, maintainability, and fidelity, ultimately hindering practical applicability and adoption.

To address this, we call for the research community to embrace "model-free verification" through network emulation. Recent technology advancements – maturation of orchestration infrastructure and vendor-provided container images – make it possible to leverage emulation to obtain a high-fidelity converged dataplane from actual router control plane code, and then apply established dataplane verification techniques to this extracted state. We prototype such a system with open-source components, and present early results showing this approach can accurately verify configurations previously untestable, paving the way for more robust, practical network verification.

CCS Concepts

• **Networks** → **Network reliability**; *Network manageability*; **Network performance evaluation**.

Keywords

Network Verification, Network Emulation, Configuration Analysis, Batfish

*Corresponding author: akrentsel@google.com

This work is licensed under a Creative Commons Attribution 4.0 International License.

HotNets '25, College Park, MD, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772380

ACM Reference Format:
Alexander Krentsel, Oliver Ye, Anthony Tafoya, Xuqian Ma, Sylvia Ratnasamy, and Anees Shaikh. 2025. Towards Accessible Model-Free Verification. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772380

1 Introduction

Maintaining network correctness remains extremely difficult for network operators today. Despite network providers' best efforts, network outages continue to occur, and have a huge impact on the many services that rely on the network for correct behavior [16, 24, 28, 39, 40].

While the root causes of outages vary, prior works [7, 18, 41] report networks are particularly vulnerable when configuration changes are made. As network size and complexity grows exponentially, so too does the frequency of configuration changes; in our network we push thousands of configuration changes per day to WAN devices for ongoing maintenance operations, network expansion, policy changes, customer-triggered changes, etc. Intent driven networking – automatically generating low-level configuration from high-level operator intent – helps avoid syntactic and so-called "fat finger" configuration bugs. However, it does not eliminate all bugs, especially related to incorrect intent.

Network verification has emerged to address this problem by formally modeling and verifying that the network's behavior satisfies some invariant, such as reachability. It can be broken into two core research directions [37]: *dataplane* verification [25, 26, 31] which checks that the set of forwarding entries in the network satisfy some correctness property, and *control plane* verification [7, 10, 12, 34] which checks that a network configuration will produce a network data plane that satisfies some intent. Control plane verification aims to verify network behavior *before* a configuration is deployed to the production network, making it more attractive for network operators as a tool for outage prevention.

Given the promise of, and research momentum behind, network verification, should we expect that verification is an integral part of a network operator's toolkit? Although reports of verification being applied at some large CSPs [1] are encouraging, broader adoption among practitioners appears to remain limited. To understand this, we surveyed 30

network engineers and interviewed ten more who manage networks ranging across enterprises, universities, small regional ISPs, multi-national ISPs, and CSPs. As we report in §2, though a majority of network engineers were familiar with network verification tooling (primarily, Batfish [7, 12] for control plane verification), only a handful of the respondents had meaningfully used verification in production.

In this paper, we ask *why* the promise of control plane verification tooling has not translated into wider adoption in real-world networks. We draw from interview and survey data, as well as our own experiences (§2) to understand the current situation – our attempts to apply existing control-plane verification tools to production-scale, multi-vendor networks have run into significant roadblocks.

We argue the core challenges are due to control plane verification's reliance on modeling of control plane behavior. Control plane verification techniques require modeling protocol behavior based on configurations, then use the model's output to reason through network behavior. But in practice, implementing a model that captures and maintains all pertinent features of the router is exceedingly difficult. As we show in §2, the real-world feature-set is large, and configuration languages are proprietary interfaces, frequently being expanded or changed, and varying from vendor to vendor. As a result, models will *always* lag in their ability to faithfully represent the actual control plane, and thus not capture real-world behavior. Indeed the most frequently stated (74%) barrier to adoption in our survey of network operators was that existing verification tools do not support the pertinent protocols or features.

We see a promising alternative path forward with the emergence of high-fidelity network emulation tools. Over the last several years, nearly all major router vendors have developed container-based virtual images [4, 5, 15, 38], which can be deployed within emulation tools. While proprietary emulators exist that can run these images [29], in recent years we've seen maturing open source network emulation infrastructure [14, 23], along with standardized vendor-independent configuration, control, and telemetry APIs [13, 32, 36]. The recent shift from VMs to containers enables scaling networks to digital twin size with reduced costs. Taken together, this broadens access for researchers and practitioners to use high-fidelity emulation in their workflows.

Based on these advancements, we argue for *model-free control plane verification* as an approach to making verification feasible on production-complexity networks. Rather than modeling control plane behavior, our approach takes router configurations and executes high-fidelity control plane emulation until convergence. We then apply traditional dataplane verification techniques to verify properties of the resulting dataplane. This provides the pre-deployment guarantees of control-plane verification, while overcoming the limitations of using control plane models to generate the dataplane. While network emulation itself is an old idea [22, 27, 29] that has even been applied with simulation and verification [34], our intent in this paper is to make the case for *model-free* verification as essential to broadening the reach of network verification, with emulation as the right enabling technology. Hence, we write this paper to encourage the research community to engage in developing a robust open-source ecosystem for network emulation, and its application to verification.

To demonstrate the accessibility of our approach, we implement a model-free verification prototype system using open source emulation and verification components. We build atop the mature Batfish verification engine and query library, which allows developers to reuse its well-documented frontend query interface to validate a variety of properties. Our early results (§5) show that our approach is able to successfully detect the impact of configuration changes unable to be parsed by Batfish, handle a wider range of features, and uncover bugs in the Batfish network model. We believe this model-free approach paves the way for more robust, practical network verification.

2 Why avoid modeling?

To understand the practical applicability of model-based control verification, we draw on both our own experience at a large CSP, as well as a survey and interviews with external network engineers. For interviews (n=10) and our survey (n=30), we recruited participants from a handful of popular network operations forums and via snowball sampling. Our survey comprised 12 questions covering background and network verification tooling experience[1]. Virtually all participants held technical network roles, across a wide range of industries including enterprise (8), ISP (7), CSP (4), government (3), and others (8). Network sizes were approximately evenly represented across small (1-50), medium (51-500), large (501-5000), and very large (5000+) by network device count. Our interviewees likewise spanned diverse sectors and geography, including university campus networks, commercial ISPs, large government scientific networks, and enterprise networks in the US, Canada, and UK. In brief, though two thirds of respondents had heard of network verification, only 30% had attempted to use it. Of the group of 10 interviewed, the main tool participants were aware of was Batfish, however only two attempted to use it in production, and zero were actively using it in their work. Those familiar with verification raised issues as we elaborate on in what follows.

Batfish. We examine Batfish [7, 12] as the reference verification tool, as it is among the most mature and widely used network verification tools, with an active Slack community

[1]Due to space limitations, we only provide a brief summary of our survey and will report on it in detail in a future publication.

with thousands of members and over 1,200 stars indicating user interest on GitHub. It was also by far the most commonly known network verification tool in our interviews.

The core network model in Batfish is called the Incremental Batfish Dataplane (IBDP) [21], which is written in Java and models an iterative exchange of information between routers and the resulting converged dataplane, for a set of the most popular router features such as IS-IS and BGP. In order to use this model, Batfish first needs to know how to configure those features in the model, so it has a "parsing" layer which extracts relevant network configuration information from the router configuration. Each vendor has their own configuration language, so there is a custom parser written for each unique configuration language.

This basic approach of modeling control plane behavior to generate a dataplane from device configurations comes with a number of drawbacks:

Coverage. The reference model only supports a subset of protocols, as modeling each protocol's behavior is very difficult: it requires deeply understanding routing protocol specs, which themselves are long and complex, with features still being added[2]. Also, for some protocols, Batfish provides no support (e.g., RSVP-TE). The protocols that *are* supported are modeled at partial fidelity, only supporting the subset of features and configuration knobs relevant to a set of target network environments. This sentiment was strongly reflected in our survey; of respondents that were familiar with network verification tooling, the most frequent (74%) of biggest barriers to adoption was that existing verification tools do not support the specific features or protocols that they use.

Correctness. For the portions of protocols that *are* supported by the model, writing the reference model implementation is tedious and error-prone for the tool developer. Over the past 7 years, there have been over 800 issues filed on GitHub for the Batfish network model and configuration parsing, for issues such as the modeled BGP route propagation differing from an observed production router [11], an L3 edge not appearing when expected [30], a valid Juniper configuration causing Batfish to crash [20], and many others. These issues are not presented as a critique of the Batfish model developers; on the contrary, we simply aim to point out how difficult it is to get the model "right" for even a subset of protocols, and how much work remains.

Maintainability. Router behavior, configuration syntax, and feature support continuously evolve on real-world routers, so must be continuously updated in the model. For example, Arista made changes to their configuration syntax in the late 2010s. The Batfish developers run continuous regression shadow-testing of their network model and parsers with real routers in the lab [12], and *still* many model and parser bugs are reported. We further observe that due to the complexity of router and network behavior, model designers often make assumptions to keep the model simple, such as assuming certain symmetries, default behaviors, etc. As fidelity needs or feature-sets evolve, if those assumptions are violated, this may require fully re-architecting the model.

Single separate implementation. A reference model of protocol behavior does not capture vendor-specific behaviors. This makes it impossible to uncover bad behavior caused by bugs or incompatibilities that appear in vendor implementations, which occur with non-trivial frequency. In our network we have observed significant numbers of issues in vendor implementations, such as a new software version that introduced an incorrect route metric selection in iBGP, or a bug where LSP deletions were not being correctly applied.

A single reference model also prevents detecting bugs that occur as a result of interplay *between* different vendor implementations. 93% of survey respondents manage multivendor networks, and so are prone to such issues. Such bugs often cannot be caught by testing done by the vendor themselves. In one such case in our network, one vendor's OS produced an unusual but valid BGP advertisement that caused another vendor's routing process to crash during parsing, leading to traffic loss and a partial network outage. In another case, poor interplay between RSVP-TE signaling timers in two vendors resulted in very slow reconvergence after a major link-cut, leading to tens of minutes of severe congestion.

Taken together, these reasons make it difficult to apply model-based verification in many real world networks. Our own attempts to apply Batfish for verification in a global, multi-vendor network over several years (2017-2021) were not successful, due to the challenges noted above. More recently (2025) we experimented with 1500 production router configurations across a number of network roles, but found that all of them failed in the parsing phase due to unsupported features in the model.

3 Emulation as an Opportunity

The traditional use case for emulation in network validation has been to develop a *digital twin* of the network on which changes can be applied and evaluated for correctness before deployment to production. Such systems may leverage models of network devices [42] or virtualized images of actual vendor device implementations [29]. Emulation has the advantage of supporting interaction using essentially the same tooling as that used to make changes on the production network. For example, when unexpected network behavior is observed, operators can SSH into emulated routers to inspect their internal state using the same router CLI tooling they use to inspect and debug production routers.

[2]For example, the BGP spec [35] has nearly 80 drafts and RFCs with proposed extensions to the standard.

However, emulation by itself is limited in that it is difficult to perform the type of exhaustive analysis that formal verification enables. Model-free verification proceeds in two steps to address this limitation. First, we use network emulation to get from network configuration and context (such as BGP advertisements) to a converged dataplane. Next, we apply traditional dataplane verification techniques on the resulting dataplane. Network emulation gives us the benefit of accurate dataplane forwarding entries without requiring manually written models and configuration parsing. Formal verification then provides valuable capabilities such as identifying specific routes that do not satisfy a desired invariant or concluding no such routes exist through exhaustive search.

From the perspective of control plane verification, this approach has a number of key benefits:

Maximum fidelity model. Vendor-supplied virtual images are the most accurate representations of device configuration and control, and are able to run production configurations similar to hardware devices. An emulation-based approach hence ensures that the resultant dataplane model reflects actual behavior.

Production interfaces and tooling. Emulated devices running vendor software support the same management and control interfaces as the production network, thus fitting naturally into existing workflows. For example, operator tools which inspect router RIBs to debug reachability issues on production routers can be applied unchanged using the emulated routers' standard interfaces (CLIs, OpenConfig, etc.). Our survey respondents echoed the importance of this benefit; 52% selected "lack of integration with existing workflows and tools" as among the biggest barriers to adoption of existing verification tooling, with nearly half rating the importance of network verification tools allowing use of familiar network operator tools at 4 or 5 out of 5.

Application to SDN and non-SDN networks. Emulated environments also support applying verification to SDN-based networks, as they support running an SDN controller and any control-plane instrumentation and programming infrastructure directly. Control inputs can be taken from current or historical production data. Thus for both SDN and non-SDN networks, a high-fidelity dataplane representation can be extracted and used by a verification engine.

Network emulation is of course not new to the research community – Emulab [22] and VINI [9], and tools such as Mininet [27] are examples of lightweight environments that provided software implementations of hosts, routers, and networks for experimentation and evaluation. However these tools did not faithfully emulate vendor router behavior.

Earlier work also reports on emulation environments with vendor router images [29], though these have not been available to the research community. And past efforts to leverage router emulation to validate correctness in large-scale networks has encountered several challenges that have been mitigated more recently as the technology has become more mature and accessible. For example, lack of virtual images from all relevant vendors made emulating multi-vendor networks challenging [42] – today, all major router vendors offer a virtualized version of their firmware. In addition, scaling to large networks is significantly improved as most vendors now support containerized versions of their images (vs. virtual machines), and emulation frameworks integrated with Kubernetes make it possible to leverage cloud platforms to more easily scale-out the emulated network [23].

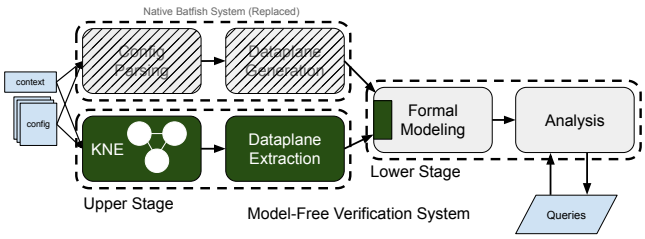

Figure 1: System diagram showing the native Batfish flow in gray, new components in dark green, and replaced components hatched.

4 Approach

Our approach builds on the opportunity that emulation provides, consisting of two core stages shown in Fig. 1. The upper stage executes the control plane emulation using virtual routers with device configurations and, optionally, additional context such as route advertisements. Once the emulation reaches a steady state, the dataplane forwarding state is extracted and provided to the bottom stage. This stage formally models the dataplane and feeds it to a verification engine, which can answer queries issued by an operator.

We aim to leverage open source components in constructing each stage of our system. In particular, we chose to build on top of Kubernetes Network Emulator [23] for emulation, and Batfish [7] for verification as it is a mature control plane verification tool with a large community of industry users and an actively maintained codebase. This enables network practitioners to use our framework as a drop-in backend for Batfish, running the same queries they may currently be familiar with. We describe each stage in turn below.

4.1 Control-Plane Emulation

The upper stage of our system takes the same inputs as Batfish, i.e. router configs, a topology file capturing links, and a set of BGP advertisements. However, rather than relying on a control plane network model, we directly emulate the

control plane behavior using Kubernetes Network Emulator (KNE) [23]. KNE is an open-source network emulator that brings up vendor-provided Router OS images, and emulates the links between their interfaces by setting up dedicated virtual networks. The control and management plane software code in the images is identical to their physical router counterparts. The main differences are in behavior related to the physical hardware, such as actual forwarding ASICs – this is simulated with vendor-provided code running on non-specialized hardware such as general purpose compute servers. KNE's ability to scale is limited only by the constraints of Kubernetes, which supports up to 150,000 pods [3]. Initializing KNE takes single to tens of minutes, depending on the network size, as pods are brought up and emulated router OSes start up; but subsequently applying new configuration to already-up routers converges much more quickly as messages are exchanged locally. After convergence, we use the vendor-independent gNMI [13] management RPC service to dump Abstract Forwarding Tables (AFTs) in the common OpenConfig [33] data models, which all vendor images now support, allowing this step to be fully vendor-agnostic.

4.2 Data-Plane Verification

The lower stage of our system consists of dataplane modeling and a verification engine that accepts verification queries. In order to maximize usability and minimize development effort, we reuse the modeling and verification engine portions of Batfish. We modified or added 3,300 lines of Java code in the Batfish backend to allow ingesting the AFTs pulled via gNMI, in place of the data plane produced by the Incremental Batfish Dataplane network model. The rest of the verification engine remains the same.

As a result, verification queries to the engine are issued through the existing Pybatfish [2] query interface in Python, which comes with a decade of refinement from real-world operator feedback, extensive documentation, and familiarity to network operators.

5 Early Results

We run the implementation of our system described in §4, with the model-free network emulation portion in a single-node Kubernetes cluster running in the cloud, and the modified Batfish verification engine running on a local machine. We use the Python Pybatfish [2] library to write and issue all verification queries. Though our method explicitly supports constructing topologies containing different vendors (which capture vendor-implementation-specific behavior), for evaluation simplicity all routers in the topology configurations below are Arista routers running cEOS Verison 4.34.0F. We report observations from our early testing.

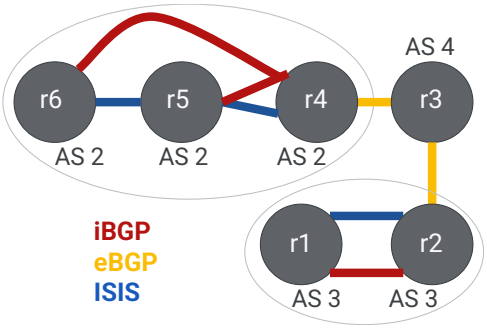

Figure 2: Diagram of our 6-node test network showing configured IS-IS and BGP communication.

```
router isis default ! Correctly parsed.
   net 49.0001.1010.1040.1030.00
   address-family ipv4 unicast
!
interface Loopback0  ! Correctly parsed.
   ip address 2.2.2.1/32
   isis enable default
   isis passive-interface default
interface Ethernet2
   ip address 100.64.0.1/31     ! Model issue #1
   no switchport
   isis enable default          ! Model issue #2
!
```

Figure 3: Router 1 Configuration Snippet.

Model-free verification can successfully uncover reachability impact. We first ensure that our system works as expected. To do this, we construct a test scenario by taking production configurations and simplifying them down to the simple setup shown in Fig. 2, containing 6 nodes that use just iBGP, eBGP, and IS-IS. The number of lines in each configuration ranges from 62-82. We then fed this configuration to our system, as well as a buggy version of the configurations where an eBGP session between R2 and R3 was taken down, and ran PyBatfish's `Differential Reachability` query across the two cases. This query type exhaustively compares network paths for all possible packets across two snapshots, and surfaces cases where the paths differ. The output correctly discovers the loss of connectivity from routers in AS3 to routers in AS2, confirming we properly integrated the emulation-derived dataplanes in the verification engine.

In our experimentation, we naturally came across scenarios that highlighted the pitfalls of modeling:

Model-based verification struggles with feature coverage. We took the same configurations used above (successfully configuring the network in Fig. 2 in emulation), and fed it to the native Batfish model-based simulation and verification system to compare its behavior. We found Batfish's

network model generation failed to recognize between 38 and 42 of lines in each configuration. Some of these lines are for enabling features that are not directly relevant to the dataplane, such as management daemon (`PowerManager`, `LedPolicy`, `Thermostat`, etc.) or configuring management services (gRPC, gNMI, SSL profiles, and more). However, others are materially relevant to the router behavior, including enabling MPLS and MPLS-TE (as these are simply not in the subset of features supported in the Batfish network model). Partial feature coverage makes it hard to trust final verification results.

Model-based verification results can be wrong or misleading. We found that assumptions baked into control plane modeling code can cause the verification results to differ from actual network behavior. Our decision to augment (vs. replace) the Batfish backend allowed us to use Batfish's built-in `Differential Reachability` question to explore differences in forwarding behavior between Batfish's native model-based simulations and our model-free emulation for identical configurations. To this end, in one of our first experiments, we constructed a simple 3-node line topology, i.e. R1 <> R2 <> R3. On each router, we assigned unique IP addresses to each interface, then enabled IS-IS as shown in the simplified configuration in Fig. 3. We then compared the results of running our model-free emulation-based system vs. the existing Batfish model-based simulation approach.

We found that the verification engine reported a difference in reachability between the two (identical) sets of configurations; Batfish's model-derived dataplane did not have reachability from R2 to R1, reporting packets to be dropped, whereas the dataplane from the actual Arista router emulation was reported to have full pair-wise reachability. After reaching out to the Batfish maintainers, further debugging revealed that Batfish's network model applied configuration in a particular order, assuming a router interface could have no IP address unless it is first configured as routed (i.e., `no switchport`). As the `ip address 100.64.0.1/31` line (issue #1 in Fig. 3) came first in our configuration, this line was simply ignored, which changed the final produced dataplane. Furthermore, `isis enable default` (issue #2 in Fig. 3) was reported as invalid syntax. As a result, the simulation's dataplane behavior meaningfully differed from the actual router's behavior of accepting this configuration.

This result shows both the difficulty and danger of relying on network models; configuration parsing and interpretation is often complex and must keep up with changes made by vendors, which can lead to misleading verification results.

Emulation performance can scale in size and complexity. To gauge emulation scalability, we ran some early tests bringing up a variety of topologies. Router container resource requirements vary by vendor; for the Arista containers we used above, each emulated router requires 0.5 vCPUs and 1 GB of RAM [23]. As such, we were able to bring up topologies of up to 60 routers on a single `e2-standard-32` 2.25GHz processor machine with 32 vCPUs and 128 GB of memory. We tested and saw successful convergence of up to 1,000 devices on a 17-node Kubernetes cluster.

To gauge convergence time with production realistic conditions, we brought up a replica of a multi-vendor 30-node portion of our network with production-complexity configurations, and inject production-recorded routes (millions from each BGP peer). We observe *convergence* time after configuration and including route injection to be approximately 3 minutes. We detect convergence to be complete once we observe the dataplane to stabilize at all routers. The one-time initial startup time of the emulation infrastructure and container booting was 12-17 minutes.

Emulation-as-a-Model fits the Network Operator tooling flow. We found an under-appreciated benefit of using emulation: in constructing network configuration for various scenarios, we could use standard network operator tools to "poke at" the control plane to inspect and debug unexpected behavior. When our initial IS-IS configuration mistakenly used incorrect syntax for Arista routers, resulting in our verification system reporting missing reachability, we were able to uncover the issue connecting to the router via SSH, and inspecting IS-IS database entries and ip route information. This echoes the sentiments of operators reported in §2. Taken together, our approach provides the powerful exhaustive search capabilities of traditional dataplane verification, with access to familiar network operator interfaces for inspection. Unlike prior work [17] which proposes integrating verification directly into the running dataplane (i.e. relying on the actual live control plane instead of control plane emulation), this allows for applying verification to any range of what-if scenarios as operators iterate before pushing changes.

6 Discussion

We acknowledge that our results are preliminary, and much remains to fully evaluate the scalability of emulation and its use in verification. In addition, a number of limitations and open questions remain, some of which are specific to our model-free approach, while others hold more generally.

Hardware bugs. As the dataplane hardware is emulated, bugs in the physical hardware will not be captured because our verification operates on the AFTs supplied to the forwarding complex, and hence do not reflect bugs in that layer. Such bugs are rare, but do occur; for example we observed a case where a hardware fault caused TCAMs on a router to not be programmed with expected forwarding entries. In another published case, a bug in the hardware-level checksum computation caused some packets to be dropped [19]. This is a problem for model-based verification as well.

Non-deterministic behavior. Some convergence behavior is dependent on the order and timing of control messages exchanged, such as the order of reservation messages in RSVP-TE [8] or the relative arrival time of BGP advertisements used as tiebreakers. Naively, one run of emulation will produce a single converged state. This limitation is even stronger in model-based verification tools like Batfish, which avoid supporting features requiring non-determinism or simplify their modeling. For higher confidence, our emulation approach can be run multiple times in parallel to produce multiple resulting dataplanes. We want to investigate further how to efficiently explore the impact of ordering.

Likewise, some bugs do not show up deterministically, or require the accrual of router state over time. An emulation run may not necessarily trigger these bugs. Model-based verification does not capture these dynamics either – exploring such implementation-level timing bugs is an open problem.

Exhaustive search across configuration scenarios. Our model-free approach operates over the same input as native Batfish, providing verification within a given network snapshot of configurations and scenario context (e.g. link statuses, external BGP advertisements). However, some network attributes of interest to operators can require reasoning over a range of possible scenarios, such as checking that the network maintains reachability in the face of any single link cut. While our system can check this, it would do so by running emulation for each new context in parallel and running exhaustive differential reachability checks across the produced data planes. This is doable for some queries but can be overly compute intensive for others such as searching any k link cuts, which grows exponentially. Such exhaustive explorations (e.g., over wide context space) are more readily supported by model-centric approaches [10].

The tradeoff we identify, however, is that relying on a network model comes with a host of problems that potentially affect the validity or practicality of the analysis. Finding *high-fidelity* and practically applicable methods of exhaustive network context search remains an open problem.

Performance verification. Recent research [6] attempts to verify bounds on performance (throughput, latency) for a constrained space of workloads. This direction is of particular interest to operators, as many bugs manifest as *performance* bugs rather than correctness bugs. Emulation does not handle comprehensively exploring a workload (e.g. demand) space the way a symbolic network performance model can. However, emulated routers do also provide dataplane emulation (with some fidelity limitations) that may provide an opportunity for examining performance for particular workloads. Furthermore, one can explore workloads on the produced dataplane model, such as checking link utilizations for a range of possible demands with the given dataplane.

Implications for the field. In this work, we aim to explore *why* production adoption of network verification seems to considerably lag research advancements. We find a key challenge to be reliance on control plane models, which are limited in fidelity and coverage, and hard to get right and maintain. This observation motivates our alternative model-free emulation-based approach using open source components.

However, despite the drawbacks we discuss in this paper, modeling still is an important technique for verification. More generally, modeling, simulation, emulation, and physical testbeds all have pros and cons in terms of fidelity, ease-of-use, and runtime/overheads. Our conclusion is that emulation is a very attractive point in the design space for making verification more practical and adoptable, one that has perhaps not received sufficient attention from the research community as full-fidelity emulation was not accessible to the community at large. We hope to motivate this direction and bring focus to recent advancements in emulation and what they enable for network researchers and practitioners – both as a tool for verification, but also more broadly as a tool for system evaluation and teaching.

Acknowledgments. We thank the anonymous reviewers and our shepherd Costin Raiciu for their insightful comments and helpful feedback. We also thank our colleagues at Google, including Neha Manjunath, Marcus Hines, Rob Shakir, Kirill Mendelev, Pranay Maddula, and others for their discussions and contributions.

References

[1] 2012. Network Verification. https://www.microsoft.com/en-us/research/project/network-verification/. Accessed: 2025-6-21.

[2] 2016. PyBatfish. https://pybatfish.readthedocs.io/en/latest/. Accessed: 2025-6-10.

[3] 2024. Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/. Accessed: 2025-6-9.

[4] 2024. vJunos-router Overview. https://www.juniper.net/documentation/us/en/software/vjunos-router/vjunos-router-kvm/topics/vjunos-router-overview-understanding.html. Accessed: 2025-5-8.

[5] 2025. IOL - Cisco Modeling Labs v2.8. https://developer.cisco.com/docs/modeling-labs/iol/. Accessed: 2025-5-8.

[6] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal Methods for Network Performance Analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 645–661. https://www.usenix.org/conference/nsdi23/presentation/tahmasbi

[7] Ari Fogel and Stanley Fung, University of California, Los Angeles, Luis Pedrosa, University of Southern California, Meg Walraed-Sullivan, Microsoft Research, Ramesh Govindan, University of Southern California, Ratul Mahajan, Microsoft Research, and Todd Millstein, University of California, Los Angeles. [n. d.]. Batfish: A General Approach to Network Configuration Analysis. *NSDI 2015* ([n. d.]).

[8] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Dr. Vijay Srinivasan, and George Swallow. 2001. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209. doi:10.17487/RFC3209

[9] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. 2006. In VINI veritas: realistic and controlled network experimentation. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA.

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168.

[11] bharmarsameer. [n. d.]. BatFish Issue 8981: Traceroute Disagreement. https://github.com/batfish/batfish/issues/8981. [Accessed 05-07-2025].

[12] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 122–135.

[13] Carl Lebsack, Marcus Hines, Paul Borman, Anees Shaikh, Rob Shakir, Wen Bo Li, et al. 2018. gNMI - gRPC Network Management Interface. https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md.

[14] Roman Dodin. 2021. ContainerLab.Dev. https://containerlab.dev/. Accessed: 2025-5-8.

[15] Roman Dodin. 2021. Nokia SR Linux goes public. https://netdevops.me/2021/nokia-sr-linux-goes-public/. Accessed: 2025-5-8.

[16] Tony Fyler. 2023. Azure Outage Disconnects Thousands. https://techhq.com/2023/01/azure-outage-disconnects-thousands. Accessed: 2024-1-30.

[17] Aaron Gember-Jacobson, Costin Raiciu, and Laurent Vanbever. 2017. Integrating Verification and Repair into the Control Plane. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets '17)*. Association for Computing Machinery, New York, NY, USA, 129–135. doi:10.1145/3152434.3152439

[18] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 58–72. doi:10.1145/2934872.2934891

[19] Chuanxiong Guo. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*.

[20] Dan Halperin. [n. d.]. BatFish Issue 8744: JunOS Parsing Crash. https://github.com/batfish/batfish/issues/8744. [Accessed 05-07-2025].

[21] Dan Halperin, Ari Fogel, Ratul Mahajan, Victor Heorhiadi, Matt Brown, Spencer Fraint, Todd Millstein, Harsh Verma, and Corina Miner. 2025. batfish/batfish. https://github.com/batfish/batfish. https://github.com/batfish/batfish

[22] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. 2008. Large-scale virtualization in the emulab network testbed. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

[23] Marcus Hines and Alex Masi. 2021. Kubernetes Network Emulator. https://github.com/openconfig/kne.

[24] Santosh Janardhan. 2021. Details About The October 4 Outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/. *Engineering at Meta* (Oct. 2021). Accessed: 2024-1-30.

[25] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 113–126.

[26] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2012. Veriflow: verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Sept. 2012), 467–472.

[27] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Monterey, California) *(Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, Article 19, 6 pages. doi:10.1145/1868447.1868466

[28] Frederic Lardinois. 2020. IBM Cloud suffers prolonged outage. *TechCrunch* (June 2020).

[29] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. (October 2017), 599–613. https://www.microsoft.com/en-us/research/publication/crystalnet-faithfully-emulating-large-production-networks/

[30] Ratul Mahajan. [n. d.]. BatFish Issue 6183: Missing L3 Edge. https://github.com/batfish/batfish/issues/6183. [Accessed 05-07-2025].

[31] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 290–301.

[32] Marcus Hines, Rob Shakir, Sam Ribeiro, Eric Breverman, et al. 2018. gNOI - gRPC Network Operations Interface. https://github.com/openconfig/gnoi.

[33] OpenConfig Project. 2015. OpenConfig. https://www.openconfig.net/.

[34] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 953–967.

[35] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. doi:10.17487/RFC4271

[36] Rob Shakir, Xiao Wang, Nathaniel Flath, et al. 2017. gRIBI - gRPC Routing Information Base Interface. https://github.com/openconfig/gribi.

[37] Ratul Mahajan Ryan Beckett. 2020. Capturing the state of research on network verification. https://netverify.fun/2-current-state-of-research/. Accessed: 2022-10-9.

[38] Whitney Sisler and Arista Networks. 2017. Arista Introduces Containerized Software for Cloud Networking - Arista. https://www.arista.com/en/company/news/press-release/2918-pr-20170307. Accessed: 2025-5-8.

[39] Richard Speed. 2021. AWS runs into IT Problems. https://www.theregister.com/2021/12/15/aws_down. Accessed: 2024-1-30.

[40] WIRED. 2019. The Catch-22 that Broke the Internet. https://arstechnica.com/information-technology/2019/06/the-catch-22-that-broke-the-internet/.

[41] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational Network Verification. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 213–227.

[42] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. doi:10.1145/3387514.3406217