Proactively Verifying Quantitative Network Policy Across Unsafe and Unreliable Environments

Yahui Li, Han Zhang, *Member, IEEE*, Jilong Wang, *Senior Member, IEEE*, Xingang Shi, *Member, IEEE*, Xia Yin, Zhiliang Wang, Jiankun Hu, *Senior Member, IEEE*, Congcong Miao, and Jianping Wu, *Fellow, IEEE*

***Abstract*— Network managers configure networks to enforce various high-level policies, and to respond to the wide range of network events (e.g., attacks, intrusions, malicious route announcements from neighbors) that may occur. It is incredibly difficult to specify these high-level policies in terms of distributed low-level configuration. These high-level policies hold only if the distributed configurations are well equipped to react to unsafe and unreliable environments (e.g., malicious route announcements, unsafe components and devices). Therefore, it is important to proactively verify whether network policies hold across continually changing environments in terms of current network configurations. State-of-the-art policy verification techniques are limited because they can check only the Boolean policies (e.g., forwarding reachability, waypoint or blackhole-freeness). However, many policy violations express themselves in quantitative ways (e.g., a link becomes overloaded). In this paper, we propose quantitative network verification (QNV) analyzing the quantitative policies of networks across unsafe and unreliable environments. QNV translates network configurations into a symbolic simulation model that captures the stable states to which the network forwarding will converge as a result of interactions between routing protocols. It then generates a logical formula matrix that describes network forwarding in the event of failures and verifies quantitative policies based on the formula matrix. We implement QNV and evaluate it on realistic and synthetic configurations. Our evaluation shows that QNV can precisely verify quantitative policies in only a few minutes, even in large networks.**

***Index Terms*— Network verification, network configuration, control plane analysis, network reliability.**

I. Introduction

TO OPERATE, maintain, and secure a communication network, network operators must grapple with low-level vendor-specific configurations to implement complex high-level network policies: e.g., specific classes of traffic (i.e, that between particular source and destination hosts) should always reach their destinations, even when some links or routers fail. However, even if these distributed low-level configurations are well checked by managers, they can still assign inappropriate packet forwarding actions responding to some environmental conditions that may occur (e.g., malicious route announcements from neighbors, a router failure caused by external attacks), which violates high-level network policies, violates customer guarantees, and impacts operator revenues [1]. Furthermore, attackers have repeatedly demonstrated their ability to compromise switches and routers [2], [3], [4], vendors have left backdoors open, and national security agencies can introduce hardware backdoors [5]. These incidents[1] force us to answer "what if" questions with respect to different environments (e.g., attacks, failures) before any new configuration is applied to a network.

Every router in a network runs distributed control logic and computes a set of forwarding rules (e.g., forwarding information base) that the router uses to forward traffic along the appropriate path. The control logic includes implementations of many protocols (e.g., Intermediate System to Intermediate System or ISIS, Open Shortest Path First or OSPF, Border Gateway Protocol or BGP), each of which uses different algorithms to exchange route advertisements with neighboring routers and compute paths. A *configuration* file (see Fig. 5) written by a network manager instructs the router on which protocols it should use, how it should select between multiple possible paths (e.g., local preference), etc. The distributed control logic, along with configurations, the states of links and routers, and outside advertisements (i.e., environments), determine the forwarding behavior in a network. A sound configuration should always be compliant with high-level policies and respond well to the wide range of unreliable environments. In the toy network shown in Fig.1, the goal of the network manager was for "users in autonomous system (AS) *A* to be able to always talk to the web server". AS *C* and AS *B* are two large provider ASes, whereas AS *A* is a customer AS. AS *A* pays provider AS *C*. A configuration plan has been reviewed and successfully deployed on routers, and the users in AS *A* can access the web server via AS *C*. One day, a router in AS *C* was attacked backdoors by sophisticated

Received 21 May 2023; revised 28 September 2023 and 9 January 2024; accepted 18 April 2024. Date of publication 5 June 2024; date of current version 7 November 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62102020 and Grant 62394322. The associate editor coordinating the review of this article and approving it for publication was Prof. Shouling Ji. *(Corresponding author: Han Zhang.)*

Yahui Li is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China.

Han Zhang, Jilong Wang, Xingang Shi, and Zhiliang Wang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Zhongguancun Laboratory, Beijing 100084, China (e-mail: zhhan@tsinghua.edu.cn).

Xia Yin and Jianping Wu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100190, China.

Jiankun Hu and Congcong Miao are with the School of Engineering and Information Technology, University of New South Wales, Australian Defence Force Academy (UNSW@ADFA), Australia.

Digital Object Identifier 10.1109/TIFS.2024.3409935

[1]Facebook experienced a major outage that left users unable to access its social network and other platforms including Instagram and WhatsApp [6], [7], for hours.

1556-6021 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

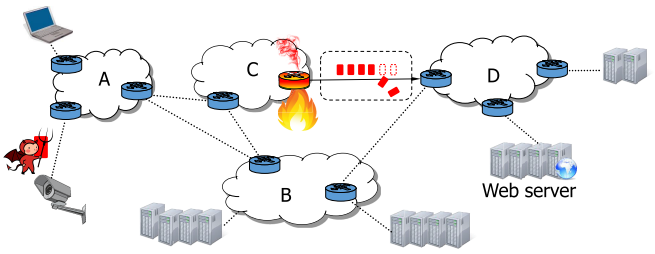

Fig. 1. An example of policy violation: a router in AS C was attacked by sophisticated malware. As soon as the router in AS C was brought down, the users in AS A cannot reach the web server anymore.

TABLE I
EXAMPLE QUANTITATIVE POLICIES

1. Given flows F with given volumes, does the load on the links in L differ by at most σ in the event of k arbitrary failures?
2. Does the end-to-end latency for all flows from A to B remain within θ in the event of k arbitrary failures?
3. Given flows F with given volumes, does link l exceed volume threshold τ in the event of k arbitrary failures?
4. Given flows F with given volumes, is there any link whose utilization will exceed δ in the event of k arbitrary failures?
5. Does traffic from A to B never traverse a path longer than n hops in the event of k arbitrary failures?

malware. As soon as the router in AS C was brought down, the users in AS A were no longer able to access the web server, thus violating the high-level policy. If the network manager of AS C configured a backup path to AS D, or AS A paid the provider AS B, this network outage could have been avoided.

Many policy verification tools (e.g., Batfish [8], abstract representation for control planes or ARC [9], efficient reachability analysis or ERA [10], Minesweeper [11], Plankton [12], symbolic router execution [13]) have been proposed to verify Boolean policies such as forwarding loops, blackholes, isolation, waypointing, and reachability, but network managers are also concerned about the *quantitative policies* (see Table I) such as latency, link load, and packet loss. However, verifying quantitative policies is non-trivial because of *the complex network control plane* and *huge failure space*. First, configurations on each router define the advertised and preferred paths to the destinations, which easily contain tens of thousands of low-level, vendor-specific primitives. The network control plane computes the forwarding path for each source–destination subnet pair, i.e., traffic class,[2] with various distributed routing protocols and complex configuration primitives. The complex cross-protocol interactions and intricate relations of protocols and configurations make it difficult to determine the forwarding behavior of each traffic class. Even more difficult is that some quantitative policies such as load violations depend on the forwarding behaviors of *multiple traffic classes*. Second, many configuration risks manifest only during link or device *failures*, with devastating impacts. For instance, the traffic load on link A may be very low when all devices in the network are up, but significantly overloaded (i.e., congestion) when one or more devices are attacked or damaged. According to a survey [9], failures are very common in networks because of various factors, e.g., attacks, human errors, and device overheating, resulting in the need for network control plane refactoring. When verifying whether quantitative policies hold in the event of arbitrary failures, the verification tool needs to check *all* failure scenarios, the number of which *grows exponentially*. This makes it impractical and unscalable to verify quantitative policies for large networks.

We address the aforementioned challenges using a framework called quantitative network verification (QNV) by employing several synergistic ideas. First, we propose an expressive and tractable *symbolic simulation* abstraction to model configurations, protocols, and their interactions. In this model, a router is viewed as a function that accepts a *symbolic route announcement* with *symbolic link state variables* as input and produces a set of symbolic route announcements for its neighbors. In each run, it determines the packet forwarding behavior in the event of *arbitrary failures*, without having to analyze forwarding paths under each failure scenario one by one. Second, we develop efficient algorithms for verifying quantitative policies based on the symbolic simulation model. These algorithms are based on a new notion of the *logical formula matrix*, without having to analyze all links and considering the load contributed by all source–destination pairs. We evaluate the utility of QNV in a range of scenarios and find that QNV can successfully verify the quantitative policies in the event of arbitrary failures, which were otherwise difficult to be verified using state-of-the-art techniques. We also evaluate the scalability of QNV and find that it can handle a network with over hundreds of routers in a minute. Our evaluations show that the model and exploration algorithms in QNV are expressive and tractable.

In summary, we make the following contributions:

- a new formal network model and its semantics, that supports model packet forwarding behavior in the event of arbitrary failures;
- efficient algorithms that verify quantitative policies based on a new notion of the logical formula matrix; and
- the implementation and experimental evaluation of QNV in a range of scenarios.

II. OVERVIEW

In this section, we closely examine quantitative property verification with network configurations and discuss some of its key challenges.

A. Violation Examples

Herein, we provide an example of real bugs encountered in real networks.

An online service provider (e.g., Microsoft's Bing) operates several data center networks. The network managers configure the network to implement their high-level policies. As shown in Fig. 2(a), with current network configurations, a large volume index file transfer is routed along the shortest path between the Los Angeles and Seattle data centers via Portland. When traffic flow uses up more than a threshold of the

[2]Following ARC [9], we use a traffic class to refer to the set of traffic flowing from one endpoint group (i.e., a set of related hosts or subnets) to another.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

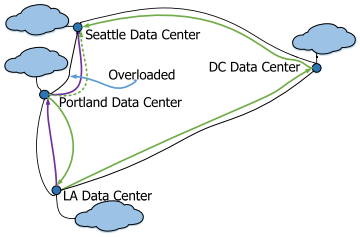

Fig. 2. A violation of latency-related property triggered by sudden increase in traffic: Blue circles are data centers, black lines are physical links between data centers, dashed green lines are previous forwarding paths for traffic flow, and solid green lines are current forwarding paths.

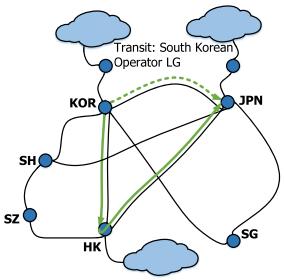

Fig. 3. A violation of latency-related property triggered by maintenance.

bandwidth between Portland and Seattle, the search traffic between Portland and Seattle will be routed along a longer path via Los Angeles and Washington by multiprotocol label switching traffic engineering (MPLS-TE); unfortunately, this greatly increases the latency of the search traffic [14]. Some studies [15] indicate that increasing latency by even a few hundred milliseconds can reduce revenue by 20%. One may argue that the traffic engineering could be modified by taking latency into account; however, the most widely available traffic engineering solution does not [14].

Another example of an online service provider is Tencent, which operates a wide area network and hundreds of data center networks that serve over one billion customers world-widely with its online services, such as QQ, WeChat Tencent Cloud, Tencent Video, Tencent Games (e.g., CrossFire, League of Legends), WeChat Pay, and QQ Mail. As shown in Fig. 3, a Tencent game service runs in its data center in Korea and serves global users. In one case, the traffic was sent along the path provided by LG, a large ISP in Korea, to users throughout the world. For some reason, network managers optimized the strategies by modifying the network configurations: the traffic to users in Korea was sent along the path provided by LG, while the traffic to users in other countries was sent via Hong Kong. Unfortunately, for the users in Japan, the traffic is sent along the long-haul submarine optical cable between Hong Kong and Japan, greatly increasing the latency of the traffic and impacting the gameplay experience of the customers.

In another example, we examine an online service provider that operates a high-performance backbone network and several data center networks. As shown in Fig. 4, router PK runs BGP, whereas router KM and router SH run OSPF. Router CHD runs BGP and OSPF and is configured to redistribute routes from OSPF to BGP [1]. An access control list (ACL)

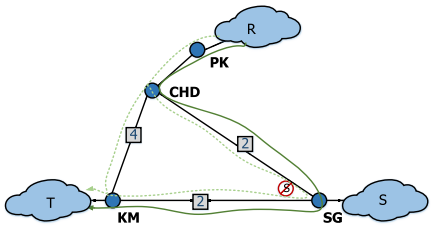

Fig. 4. A violation of latency-related property triggered by link failure.

is configured at router SH to block $S \rightarrow T$ traffic for a security policy. Router CHD is configured to support load balancing for the traffic from R to T. All links in the network have a capacity of 100 Gbps. One day, the link CHD $\rightarrow$ KM failed for a while, and the traffic for classes $S \rightarrow T$ and $R \rightarrow T$ started losing packets. Afterward, the network managers analyzed the incident. The traffic for classes $R \rightarrow T$ and $S \rightarrow T$ are 80 Gbps and 30 Gbps, respectively. When no links have failed, the traffic from $S \rightarrow T$ flows through the path SH $\rightarrow$ KM. The traffic from $R \rightarrow T$ is load-balanced at router C; 40 Gbps traffic is sent through PK $\rightarrow$ CHD $\rightarrow$ KM and PK $\rightarrow$ CHD $\rightarrow$ SH $\rightarrow$ KM. When CHD $\rightarrow$ KM failed, the traffic on SH $\rightarrow$ KM was 110 Gbps, exceeding the capacity of that link. Given a traffic matrix (TM), if the network managers can inspect whether certain links will be overloaded in the event of a single link failure based on the current network configuration, then this incident could be avoided.

These configuration risks are very common in data centers, wide area networks, and internet service provider networks [1]. The obscure vulnerabilities in routing configurations are the major reason for violations of the reachability or performance intents set by network managers [16]. A possible way to detect these performance problems, such as latency inflation, is to send probe packets in runtime. By contrast, QNV can detect problems *proactively* before customers notice them. Further, these *risks* in configurations are always latent and manifest only under *certain environments*. It seems preferable for the network manager to directly ask whether the current network configuration fulfills their high-level reachability or performance intents. If intent violations are detected, network managers could generate minimal configuration repairs (e.g., changing a set of devices or adding configuration snippets) to avoid devastating effects. This is the capability we seek to provide.

B. Router Configuration

We can think of a router as a special-purpose computer. It has its own operating system, which is called the internetwork operating system (IOS), together with files and file systems. The information of the router is stored in a device called the non-volatile random-access memory (NVRAM), whereas the IOS images are stored in the flash (lowercase). The configuration file that describes the current running state of the router is called running-config. Fig. 5 provides a simplified running-config from a Cisco router. Briefly, lines $1-2$ indicate the IP address on interface Ethernet0. Line 4 defines a static route that assigns the value 10.168.4.0 255.255.255.0 to

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

```
1 interface Ethernet0
2 ip address 10.16.0.0/31
3
4 ip route 10.168.4.0 255.255.255.0 10.168.2.0
5
6 router ospf 1
7 redistribute static metric 20 subnets
8 distance 70
9 network 10.168.42.0 0.0.0.255 area 0
10
11 bgp router 1
12 redistribute static
13 neighbor 10.16.0.1 remote-as 2
14 neighbor 10.16.0.1 route-map M out
15
16 ip community-list standard cm permit 1:1 2:1
17 ip prefix-list pf permit 10.168.2.0/24
18
19 route-map M permit 10
20 match community cm
21 match ip address prefix-list pf
22 set local-preference 200
23
24 route-map M permit 20
25 set metric 80
```

Fig. 5. Example router configuration.

the destination address of the route and the value 10.168.2.0 to the IP of the next-hop router. Lines 6−9 indicate that the router is running OSPF. Lines 7−8 show that routes from static routes and subnets are redistributed into OSPF, and that the administrative distance of redistributed routes is assigned the value 70. Line 9 determines that the router will activate the interface with the IP address 10.168.42.0 0.0.0.255, and that the interface belongs to area 0. Lines $11-14$ indicate that the router is running BGP. Line 12 shows that static routes are redistributed into BGP. Lines $13-14$ define the IP address of the neighbor, their AS number, and an outbound route map. Line 16 establishes a community list that permits announcements for any prefix with the community $1:1$ or $2:1$, and line 17 defines a prefix list that permits announcements for any prefix that matches the first 16 bits of 10.168.2.0. Lines $19-25$ identify the details of the route map M. Lines $19-22$ define the route map entry with sequence 10 and set the local preference attribute to 200 for the prefixes that match with the community list cm and prefix list pf. Lines $24-24$ define the route map entry with sequence 20 and set the multi-exit discriminator (MED) value to 80 for any prefix. Route map entries are evaluated in order starting with the lowest sequence number and ending with the highest sequence number.

Even in this small configuration snippet, some of the challenges for formal analysis with configurations quickly become apparent. On one hand, the *configuration instruction set* is extremely *complex*. Network managers today still use these complex commands written in low-level configuration language to achieve high-level policies. For instance, Cisco IOS [17] contains over $14,000$ configuration commands and over 200 for BGP alone. On the other hand, the *semantics* of the protocols are *not explicit in the configuration*. Instead, they are scattered throughout the configuration and the protocol request for comments (RFC). How the router selects a route is captured partially through distributed configurations and partially through the codes written following the RFC.

C. Challenges

The *complex configuration instruction set*, *incomplete semantics*, and *distributed nature of routing* make the task of formal verification with configurations a challenging problem. However, that is not the sole challenge in verifying quantitative policies. First, some of the quantitative policies depend on the behavior of *all traffic classes* in the network rather than only a portion of traffic classes[3]. Analyzing whether a network satisfies certain quantitative policies, such as whether the network links are overloaded, requires more than determining the forwarding behavior of a pair of traffic classes (i.e., a source–destination subnet pair) with configurations. We have to construct an effective network model to infer the *collective load imposed by all traffic classes* based on low-level router configurations. Second, in many cases, the impact of configuration vulnerabilities manifests only *in the event of failures*. For instance, some of the links in the network will be overloaded if devices or links fail. If the network satisfies a certain quantitative property in the event of k failures, it signifies that the property holds on $\binom{k}{n}$ data planes for every possible failure scenarios, where k is the number of failed links, and n is the number of links in the network. The number of failure scenarios *grows exponentially* with number of failed links, and therefore, enumerating all failure scenarios can be prohibitively expensive.

D. Key Ideas

Symbolic execution [18] is a way to abstractly execute a program by making the inputs symbolic. When the symbolic executor encounters a conditional branch (e.g., if–else statement), it executes each branch and updates the path condition, which is a set of constraints encoding the branching decisions during the execution. One abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. The execution "returns" a result that is expressed in terms of symbolic constants that represent those input values. We borrow the symbolic concept from the program community and propose *symbolic simulation* to address the scalability challenge caused by failure scenarios. Rather than generating data planes one by one for every possible failure scenario, it encodes the failures (i.e., link states) as symbolic variables and *symbolically* executes the network model. With this symbolic simulation, we can obtain the stable states (i.e, extended FIB with *symbolic variables*, symbolic data plane) to which the network forwarding will converge in the event of failures.

Another idea is to use a logical formula matrix to address the scalability challenge of reasoning the behavior all traffic classes. Each item in the data plane matrix consists of the refined route attributes with logical formulas that indicate whether the packets from a router can reach the gateway router of the destination subnet. It checks all the route attributes in the matrix, and computes the sub-collections, each of which consists of route attributes corresponding to the same failure scenario. It checks whether the load on a link exceeds the threshold with these sub-collections, rather than computing the behavior for all traffic classes one by one with the symbolic data plane.

[3]As mentioned in ARC [9], a traffic class represents the set of traffic flowing from one endpoint group–a set of related hosts, subnets, etc.–to another.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

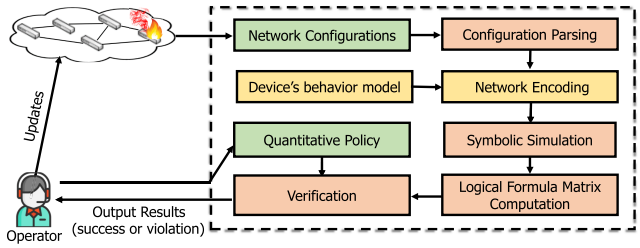

Fig. 6. Workflow of QNV. QNV obtains the running router configuration files in the network through secure shell (SSH) protocol or automated tools, obtains the properties input by the network managers, and verifies whether the network fulfills these properties across various environments.

As shown in Fig. 6, QNV loads the target configuration to be verified from the network and the property to be verified from the network manager. QNV offers verification services for network managers to check whether a set of configurations fulfills some particular properties across unsafe and unreliable environments, facilitating network managers to repair and upgrade the network configuration. Then, it loads the behavior models of the devices and feeds them with the target configuration to generate the target network model (detailed in §III). After that, it symbolically executes the network model (§IV-A), computes the logical formula matrix (§IV-B), and answers verification questions set by the network manager (§IV-C).

III. Network Model

To determine and analyze the packet forwarding behavior in a network, we first define a network model and its routing and forwarding semantics. Although research studies have been conducted on verification, the network models demonstrated in in these studies suffer from critical limitations. Some focus on a limited set of routing protocol features (e.g., ARC [9]). They are thus unable to capture the behavior of networks that use multiple routing protocols and sophisticated features. On the other hand, the model in Batfish [8] simulates the entire process of route propagation and convergence on the control plane, which makes it prohibitive to iteratively use Batfish to analyze the impact of many environments; Minesweeper [11] models the configuration logic of an entire network as a single, big logical formula with diverse constraints, which is impractical to solve even when using a modern satisfiability modulo theories (SMT) solver. Therefore, we propose the stable forwarding problem instance (SFPI), a generic model of routing mechanisms and the network on which it runs. SFPI encodes route attributes symbolically, and describes runtime routing behavior in terms of local processing of routing messages, as configurations do, which makes it easy to infer packet forwarding behavior across many environments.

A. Network Encoding

As shown in Table II, an SFPI is a tuple $\mathcal{N}_d^s = (V, I, E, R_\perp, A_\perp, \oplus, \texttt{init}, \texttt{trans}, \texttt{filter})$, where (V, I, E) is a network topology with a set of nodes V, a set of ports $V.I$, and a set of links E; and $e = (v.i, \bar{v}.\bar{i})$ is a link between port $v.i$ and port $\bar{v}.\bar{i}$, where $v.i \in V.I$, $\bar{v}.\bar{i} \in V.I$, and $e \in E$.

TABLE II
Relevant Definitions for SFPI

$\mathcal{N}_d^s = (V, I, E, R_\perp, A_\perp, \oplus, \texttt{init}, \texttt{trans}, \texttt{filter})$			
V			Nodes
I			Ports
E	$\subset$	$V.I \times V.I$	Links
$R_\perp$	$=$	$R \cup \{\perp\}$	Route attributes
$A_\perp$	$=$	$A \cup \{\perp\}$	Packet attributes
$\oplus$	:	$R \times R \to R$	Compare function
$\texttt{init}$	:	$V \to R$	Init function
$\texttt{trans}$	:	$E \times R \to R$	Transfer function
$\texttt{filter}$	:	$E \times A \to A$	ACL function

TABLE III
Main Fields of a Route Attribute r

Variable	Description
prefix	Prefix
length	Prefix length
ad	Administrative distance
met	Protocol metric
rid	Neighbor router ID
lp	BGP local preference
med	BGP multi-exit discriminator (MED) attribute
comm	BGP community
cond	Vaild condtions

Meanwhile, R is a set of *attributes* that describe the fields of routing messages processed by protocol instances. Different protocols use different kinds of routing messages. For instance, BGP messages include a destination prefix, an AS path, and other optional data; whereas the Routing Information Protocol (RIP) messages include a destination prefix and hop count. Table III outlines the main fields of a route attribute r, where $r \in R$. The measure ad is used to determine which protocol instance is used for forwarding, when multiple protocol instances offer a route to the same prefix. The parametermet is a protocol-specific measure of the route attribute (e.g., hop count for RIP, path length for BGP, path cost for OSPF). The measure rid is used to break ties among equally good routes. The parameter lp is used to manipulate the best outbound path, when BGP protocol instance learns more than one route to the same prefix. Other protocol-specific attributes such as the BGP multi-exit discriminator (med) and BGP community comm are also included in each route attribute. Finally, every route attribute contains one special field cond, which models the conditions that make this route attribute valid. When cond is true, the remaining contents of the attribute are meaningful. For instance, cond can be a logic formula that is composed of binary link aliveness variables. When cond$=s_{(v.i,\bar{v}.\bar{i})} \wedge \neg s_{(v.i,\hat{v}.\hat{i})}$, it indicates the route attribute is valid only when the status of the link $(v.i, \bar{v}.\bar{i})$ is up and the status of the link $(v.i, \hat{v}.\hat{i})$ is down.

Additionally, $\oplus$ is a compare function, which compares the attributes of two different routes and chooses the better one, where "better" is defined from the perspective of the routing decision procedure. If attribute $\bar{r} \oplus \hat{r} = \bar{r}$, where $\bar{r} \in R$ and $\hat{r} \in R$, then $\bar{r}$ is more desirable. With the $\oplus$ function, routes can be ordered lexicographically, first by administrative distance, second by local preference, and third by the length of their paths.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
MAIN FIELDS OF A PACKET ATTRIBUTE a

Variable	Description
dstIP	Destination IP address
srcIP	Source IP address
dstPort	Destination port
srcPort	Source port
protocol	Packet protocol

Meanwhile, `init` is the init function, which models the initial route attribute of each node in the network. $\texttt{init}(v)$ describes the initial route attributes at node v.

Finally, `trans` is the transfer function that models how route attributes are modified or dropped as they are propagated through the network. Given an edge $(v.i, \bar{v}.\bar{i})$ and a route attribute r from $\bar{v}$'s neighbor v, `trans` determines what new route attribute is received at the current node $\bar{v}$. The actions in `trans` are customized using routing protocols and routing configurations, including user-defined policy (e.g., output filtering on neighbor, input filtering of the current node), route aggregation, and route selection, among others.

We are interested in analyzing the packet forwarding in the network data plane, and not only the route propagation in the control planeplane. Therefore, we define the attributes of packets A. Table IV shows the main fields of a packet attribute a, where $a \in A$. After a router receives a packet, there may be multiple routes matched with that packet; the router uses the longest matching prefix to decide which route to use. We use the function `fbm`: $R \rightarrow A$ to capture the dependence between the route attribute and packet attribute, where destination prefix of the route attribute applies to the destination IP of the packet attribute. If $a = \texttt{fbm}(r)$, the first r.length bits of the prefix r.prefix and destination IP a.dstIP are common.

Even if the control plane protocols decide how to forward packets, the final packet forwarding can be restricted by ACLs configured on router interfaces. Given an edge $(v.i, \bar{v}.\bar{i})$ and a packet attribute a from the neighbor $\bar{v}$ across the edge, the function `filter` determines what new packet attribute is received at the current node $\bar{v}$.

B. Network Solution

In well-designed networks, the network will eventually converge to a *stable state*, where each node selects a single best route for each destination among the ones received from neighbors, and will not improve the routes that it has currently selected. We define the *solution* for the SFPI to model the stable state of the network. Formally, control plane solution $\mathcal{F} : V \rightarrow R$ encodes the final route (attribute) for each destination, and data plane solution $\mathcal{L} : V \rightarrow A$ encodes the final packet attribute.

The solution $\mathcal{F}(v)$ maps node v to a final route chosen to forward the traffic and satisfies the following constraints:

$$\begin{aligned}
&\mathcal{F}(v) \\
&= \texttt{sort}([\texttt{init}(v), r_0, r_1, \ldots, r_n], \oplus) \\
&\quad \times \texttt{where } r_0, r_1, \ldots, r_n \in \texttt{attr}_{\mathcal{F}}(v), \\
&\texttt{attr}_{\mathcal{F}}(v) \\
&= \{r | (r, e) \in \texttt{choices}_{\mathcal{F}}(v), e = (\bar{v}.\bar{i}, v.i)\}, \\
&\texttt{choices}_{\mathcal{F}}(v) \\
&= \{(r, e) | e = (\bar{v}.\bar{i}, v.i), r = \texttt{trans}(e, \mathcal{F}(\bar{v})), r \neq \perp\}, \\
&\texttt{fwd}_{\mathcal{F}}(v) \\
&= \{e | (e, r) \in \texttt{choices}_{\mathcal{F}}(v), e = (\bar{v}.\bar{i}, v.i), r = \mathcal{F}(v)\},
\end{aligned}$$

where $\texttt{choices}_{\mathcal{F}}(v)$ defines the routing choices available from its neighbors on node v is computed by applying the transfer function f over the solution $\mathcal{F}(\bar{v})$ of each neighbor and edge $e = (\bar{v}.\bar{i}, v.i)$. Meanwhile, $\texttt{attr}_{\mathcal{F}}(v)$ are the available routes in $\texttt{choices}_{\mathcal{F}}(v)$. The control plane solution of node v can then be modeled as a combination of the available routes provided by its neighbors and its initial route r_d. The solution $\mathcal{F}(v)$ is modeled as the winner in $\texttt{attr}_{\mathcal{F}}(v)$ by applying the compare function $\oplus$. Finally, $\texttt{fwd}_{\mathcal{F}}(v)$ models the edges e that are used to learn the best route.

$$\begin{aligned}
&\texttt{choices}_{\mathcal{L}}(v) \\
&= \{(a, e) | e = (v.i, \bar{v}.\bar{i}), (\bar{v}.\bar{i}, v.i) \in \texttt{fwd}_{\mathcal{F}}(\bar{v}), \\
&\qquad a = \texttt{filter}(\texttt{fbm}(\mathcal{L}(\bar{v})), v.i)\}
\end{aligned}$$

We use $\texttt{choices}_{\mathcal{L}}(v)$ to refer to the packet forwarding choices from the neighbors of v. Specifically, $(\bar{v}.\bar{i}, v.i) \in \texttt{fwd}_{\mathcal{F}}(\bar{v})$ refers to the edges used to learn the best routing choices on $\bar{v}$. The packet passes through the network along the opposite that in which the routing information flows. The set choices $\texttt{choices}_{\mathcal{L}}(v)$ can then be obtained by applying the ACL function t on $\mathcal{L}(\bar{v})$:

$$\mathcal{L}(v) = \{a | (a, e) \in \texttt{choices}_{\mathcal{L}}(v)\}.$$

The data plane solution of node v can then be modeled as available packet attributes in $\texttt{choices}_{\mathcal{L}}(v)$.

IV. VERIFICATION

A. Symbolic Simulation

Whereas an SFPI defines the basic operations routers perform to modify routing messages and packets, it does not indicate how to compute the SFPI solution. This computation is complicated by the fact that the transfer functions are configuration-specific and routing messages are exchanged asynchronously. In this section, we propose the symbolic simulation approach, which enables fast property verification in the event of arbitrary failures. By encoding the *link state* as a *symbolic variable* and deriving the conditions of all route attributes, it computes all possible outcomes of the distributed control plane computation (i.e., the SFPI solution) in each run. With these outcomes, one can then analyze the complete set of possible final states and verify if the property of interest holds.

Computing SFPI Solution. Algorithm 1 outlines the key steps used to compute the solution $\mathcal{F}$. The first step is to initialize a priority queue q, which is used to store the nodes to be processed (line 1). For each node v in the network, the solution $\mathcal{F}$ is initialized, the tuple $\{v, \texttt{init}(v)\}$ is put into $\texttt{received}(v)$, and the queue q is populated with the node (lines 2–5). Next, nodes from the front of the queue

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Symbolic Simulation

```
   Function: Simulate(V, E, init, trans, ⊕)
1  q ← {};
2  foreach v ∈ V do
3      F(v) ← init(v);
4      received(v) ← received(v) ∪ {v, init(v)};
       /* Routes received at node v.                    */
5      q ← q ∪ {v};
6  while notEmpty(q) do
7      v ← pop(q);
       /* Propagate v's route.                          */
8      foreach u ∈ neighbor(v) do
9          new ← trans((v, u), F(v));
10         if v ∈ received(u).getKeys() then
               /* Is there a stale route?               */
11             route ←sort({received(u).getValues()}, ⊕);
12         else
13             route ← sort({F(u), new}, ⊕);
14         if notEqual(route, F(u)) then
15             F(u) ← route;
16             q ←q ∪ {u};
17         received(u) ← received(u) ∪ {v, new};
   Function: trans((v, u), F(v))
18 new ← φ;
19 foreach r ∈ F(v) do
20     r ← transEngressPolicy(r, v, u);
21     if valid(r.cond) then
22         r̂ ← generate(r, r.cond∧ stateUp(v, u));
23         r̂ ← transIngressPolicy(r̂, u, v);
24         new ← new ∪ r̂;
25 return new;
   Function: sort (route, ⊕)
26 route ← sortbyPriority(route, ⊕);
   /* Perfer higher LocalPref, shorter ASPathLen,
      lower origin type, lower MED, etc.             */
27 better_cond ← φ;
28 foreach r ∈ route do
29     r.cond ← r.cond ∧ ¬better_cond;
       /* Update r.cond by negating the conditions of
          all higher-priority routes.                */
30     better_cond ← better_cond ∨ r.cond;
```

q are repeatedly popped and processed while the queue is not empty (line 6−15). For a node v, the current routes on v is propagated to all of the neighbors of v (lines 6−7). For each neighbor u, the solution $\mathcal{F}(u)$ is updated (lines 8−14). New routes received on node u are computed by applying the transfer function on $\mathcal{F}(v)$ (line 9). Next, the node u is checked for whether it has previously received information from node v with the function `received`(u) (line 10). If node u has received information from node v, it indicates that the solution on node u contains stale information from node v. The solution $\mathcal{F}(u)$ is updated using all received messages, including the new routes from v (lines 10−11). Otherwise, the new information is merged with the current solution $\mathcal{F}(u)$ by the `sort` function (lines 12−13). For routes to multiple destinations, one can run the simulations in parallel.

Exporting Routes and Importing Routes. The `trans` function shows how to update $\mathcal{F}(v)$ along the edge (v, u) (lines 19–29). It first filters or modifies for each route r in $\mathcal{F}(v)$ according to the egress policies of node v (lines 20−22). If r passes the egress policies of v and r.cond is valid, it updates the conditions of r by r.cond← r.cond∧ `stateUp`(v, u) and generates a new route $\hat{r}$ with r for node u (lines 27–29). Finally, it filters or modifies $\hat{r}$ according to the ingress policies of node u.

Ranking Routes: The `sort` function lists the key steps to ranking and updating the received routes (lines 26−30). It ranks the received routes according to their priorities in descending order (line 26). The priorities of these routes are determined by the protocol RFC. For instance, when there are multiple routes for the same prefix, it prefers a route with higher local preference, shorter AS path length, lower origin type, lower MED, etc. With the transfer function, we see that each route in $\mathcal{F}(v)$ is applied to any export policies and forwarded to the neighbors of the node. Notice that the forwarding of a lower-priority route depends on high-priority routes. The forwarding is valid only when the high-priority routes are invalid. Therefore, it adjusts the condition of the lower priority route by "anding" the negation of the conditions of the higher-priority routes (lines 27−30).

With the control plane solution, we can verify whether the route to a given subnet can reach a node v in the event of k failures. By inspecting the solution $\mathcal{F}(v)$, we can obtain the candidate route attributes r_1, r_2, r_3, …, r_n for the subnet. The conditions that make at least one route exist is r_1.cond∨r_2.cond∨ r_3.cond∨…∨r_n.cond. By leveraging a logic solver (e.g., Z3 solver [11]), we can easily obtain the minimum number of False variables to make that formula False.

B. Logic Formula Matrix Computation

Our goal is to verify quantitative policies in the event of arbitrary failures. Some of these quantitative policies are related to multi-flows in the network. For instance, we have to verify whether the links in the network are overloaded. Answering these questions requires analyzing the packet forwarding behavior under possible link failure scenarios.

1) Control Plane Matrix: We define a *control plane matrix* CP, which describes all possible routes that can materialize (i.e., become the best route) when links fail, and the corresponding failure scenarios. Each item cp_{ij} in CP consists of the route attributes $\mathcal{F}(v_j)$ on node v_j to the gateway router v_i of the destination subnet. The control plane matrix provides a set of routes on each node, but it indicates nothing yet about how packets are forwarded. Because of the longest-prefix matching and ACL rules, "the route of a subnet can reach node v" does not necessarily imply "a packet can reach the gateway router of the subnet from v".

2) Data Plane Matrix: To obtain information on the packet forwarding reachability about the data plane, a simple way is to test whether the destination prefixes of the route attributes on each node apply to the destination IP of the packet of interest and update the forwarding condition of the packets according to longest-prefix matching and the ACLs. However, there may be an intractably large number of tests for the entire network, making the computation unscalable. To address this, QNV infers the possible forwarding paths with the control plane solution and identifies the potentially blocked packets because of longest-prefix matching and the ACLs on these paths by refining the control plane solution. In this way, we can

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Data Plane Matrix Generation

```
Function: generateMatrix(N, CP, DP)
1  foreach cp ∈ CP do
2      foreach r_i ∈ getEntries(cp) do
           /* (r_i.prefix, r_i.length, r_i.cond).          */
3          foreach r_j ∈ extractEntries(getCond(r_i)) do
               // each node on r_i's propogation path
4              if longerPrefix(r_j, r_i)= true then
                   /* long bit match                       */
5                  r_k ← splitEntry(r_i);
6                  r_i ← updateEntryPref
                   (applyBDDs(r_i, r_j, −));
                   // (r_i.prefix', r_i.length', r_i.cond).
7                  r_k ← updateEntryCond (r_k.cond ∧ ¬r_j.cond);
                   // (r_i.prefix, r_i.length, r_i.cond').
8                  cp ← updateCp(r_i, r_k);
9      DP ← insertToDP(tanspose(cp));
10 foreach dp ∈ DP do
11     foreach r_i ∈ getEntries(dp) do
12         foreach L ∈ getACLs(getCond(r_i), N) do
               /* each ACL.                                */
13             foreach l_i ∈ L do
                   /* each rule in an ACL.                 */
14                 if actionDeny(l_i)= true then
15                     p_i ← predicates(l_i);
16                     r_i ← updateEntryPref
                       (applyBDDs(r_i, p_i, −));
                       // (r_i.prefix', r_i.length', r_i.cond).
17             if getEntryPref(r_i) = false then
18                 break;
19         dp ← updateDp(r_i);
```

construct a *data plane matrix* DP, in which eachitem dp_{ij} consists of the refined route attributes, which indicate whether the packets from router v_j can reach the gateway router v_i of the destination subnet.

Algorithm 2 outlines the main steps to computing the data plane matrix. For each route attribute r_i in each cp (lines 1−2), the effect of longest-prefix on r_i is first inferred by refining r_i.prefix and r_i.cond (lines 3−8). With r_i.cond, the nodes on the propagation path of r_i are inferred, and the route attributes on these nodes are loaded ((line 3). For each route attribute r_j, if it has the longer matching prefix (line 4), r_i.prefix is updated by computing the difference set of r_i.prefix and r_j.prefix (lines 5−6), and a new route attribute r_k whose condition is the r_i.cond "anding" the negation of r_j.cond is generated (line 7). Next, the data plane matrix DP is initiated by transposing the control plane matrix (line 9). For each attribute r_i in each dp (lines 10−11), the effect of ACLs on r_i is processed by refining r_i.prefix and r_i.cond (lines 12−18). With r_i.cond, the nodes on the propagation path of r_i are inferred, the ACLs L on these nodes are loaded (line 12). For each rule l_i in each ACL (line 13), if the action of l_i is "deny", r_i.prefix is updated by computing the difference set of r_i.prefix and p_i, where p_i is the constraint that must be satisfied to trigger an action from l_i (lines 14−19).

Each ACL rule is specified by a predicate (that guard input) and an action. The predicate of an ACL rule can be represented by a binary decision diagram (BDD). A BDD is a rooted, directed acyclic graph, which can compactly represent a Boolean function. Each node represents a Boolean variable used to represent the prefix fields. A solid arrow denotes that the value is true, whereas a dashed arrow signifies that the value is false. There are two leaf values: 0 and 1, which represent false and true, respectively. Any path from the BDD root to 1 represents a valid input–output relation. This turns checking the intersection of the ACL and prefix (line 11) into an $\mathcal{O}(n)$ operation after their BDDs are constructed.

C. Policy Verification

As noted earlier, network configurations always work correctly in the absence of failures, and some risks manifest only during failures. For each of the following quantitative policies, we show how to verify whether it holds in the event of k failures.

1) Hop Count: Network managers often want to ensure that a specific traffic tc is completed within n hops in the event of k failures. As shown in Algorithm 3, for the traffic tc to be verified, the matched entry r_i from the data plane matrix is extracted, and a BDD is constructed using r_i.cond (lines 2−3). The length t of the shortest path from the root node to terminal node false in the BDD is then computed. If $k > (t-1)$, we determine that the reachability for tc for k failures does not hold (lines 4−6). Otherwise, it checks the paths from the root node to the terminal node true in the BDD (lines 7−9). If there exists a path whose length is longer than n, we determine that the property of n hops for k failures does not hold (lines 10−13).

2) Latency: In many settings, it is desirable to guarantee that the average end-to-end latency for all flows between two endpoints is within a few milliseconds. Algorithm 3 shows how to verify whether traffic tc is completed within θ for k failures (lines 14−22). The computation is similar to the hop count verification. If there exists a path p whose latency $\sum_{e\in p}\texttt{getLatency}(e)$ is larger than θ, we determine that the property for k failures does not hold, where $\texttt{getLatency}(e)$ returns the propagation delay of link e (line 21). The end-to-end latency is the sum of the transmission delay, propagation delay, queuing delay, and processing delay. If the propagation delay between two endpoints is larger than the required latency guarantee, then clearly the end-to-end latency will not be satisfied.

3) Link-Load Violation: Link-load violations occur when traffic passes through a link whose utilization exceeds its capacity. Unlike the hop count property, the link-load violation property depends on the forwarding behavior of multiple flows. To check whether the load on link e exceeds the threshold τ for k failures, it requires exploiting the forwarding behavior of multiple flows for every possible failure scenario. Instead of trying to generate every possible failure scenario, QNV splits the collection of route attributes in DP into sub-collections according to the conditions of the route attributes for the route attributes in each sub-collection to correspond to a possible failure scenario. As shown in Algorithm 3 (lines 23−30), iterations are performed over each item in DP. If the condition of route attribute r_i and the condition of route attribute r_j have intersections, it signifies that r_i and r_j could coexist under a possible failure scenario (lines 26−27). By checking all

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3 Verification

```
   Function: verifyHops(DP, tc, k, n)
   /* k,n.                                                    */
1  flag ← false;
2  r_i ← matchedEntry(tc, DP);
3  bdd ← initializeBdd(getCond(r_i));
4  t ← shortestPathlength(bdd, false);
   /* the shortest path from the root node to
      terminal node false                                     */
5  if k > (t − 1) then
6      flag ← true;
7  else
8      P ← getPaths(bdd, true);
9      foreach p ∈ P do
10         if getlength(p) > n then
11             flag ← true; break;
12 if flag = true then
13     ex ← ex ∪ {getCounterExam(tc, bdd)};
   Function: verifyLatency(DP, tc, k, θ)
   /* k,c.                                                    */
14 flag ← false;
15 r_i ← matchedEntry(tc, DP);
16 bdd ← initializeBdd(getCond(r_i));
17 if k > (shortestPathlength(bdd, false) − 1) then
18     flag ← true;
19 else
20     foreach p ∈ getPaths(bdd, true) do
21         if ∑_{e∈p} getLatency(e) > θ then
22             flag ← true;break;
   Function: verifyLinkload(DP, e, τ, k)
23 flag ← false;
24 foreach r_i ∈ getEntries(DP) do
25     foreach r_j ∈ (getEntries(DP) − r_i) do
26         if applyBDDs(r_i.cond, r_j.cond, ∧, e, k) = false then
27             R ← updatePartitions(R, {r_i, r_j});
               /* R^0, R^1, ..., R^j;                         */
28 foreach R^j ∈ R do
29     if ∑_{r_i∈R^j} getLoad(r_i.prefix, r_i.length) > τ then
30         flag ← true;break;
   Function: verifyBalancing(DP, e_i, e_j, δ)
31 flag ← false; b ← 0;
32 L^{e_i} ← compLinkLoad(DP, e_i);
   /* l_0^{e_i} : (load, bdd);                                */
33 L^{e_j} ← compLinkLoad(DP, e_j);
34 foreach l_p^{e_i} ∈ L^{e_i} do
35     foreach l_q^{e_j} ∈ L^{e_j} do
36         if applyBDDs(l_p^{e_i}.bdd, l_q^{e_j}.bdd, ∧) ≠ false then
37             b ← updateThreshold (l_p^{e_i}.load, l_p^{e_j}.load, b);
38 if b > δ then
39     flag ← true;
```

the route attributes in DP, the sub-collections $R^0, R^1, \ldots, R^j$ are obtained, each of which consists of route attributes corresponding to a possible failure scenario. Next, iterations are performed over each sub-collection R^j. If load $\sum_{r_i \in R^j} \texttt{getLoad}(r_i.\text{prefix}, r_i.\text{length}) > \tau$, a case is reported, stating that the load on link e exceeds the threshold τ (lines 28−30).

4) Load Balancing: Load balancing is designed to evenly distribute network traffic to prevent load violations. In many settings, managers want to ensure that the load on link e_i and link e_j differs by at most δ. As shown in Algorithm 3 (lines 31−32), the load L^{e_i} on e_i is computed. Each $l^{e_i} \in L^{e_i}$ is a pair (load, bdd), where bdd refers to the topology conditions corresponding to a possible failure scenario. Similarly, the load L^{e_j} on e_j is computed (line 33). Next, for each pair of l^{e_i} and l^{e_j}, a different value b of the load is calculated. If $b > \delta$, a property violation is reported (lines 34−39).

D. Optimizations for Scalability

1) Incremental Updating: Algorithm 1 shows how to compute the control plane solution, which enables property verification in the event of arbitrary failures. In this simulation process, each node will update its current best route attributes by applying the transfer function to all of its current routes from neighbors. However, this is not always necessary. Suppose a node u received a route (old) from its neighbor v, and node v now sends a new route (new) to node u. If old⊕new=new, it suffices to insert new into the existing solution rather than recomputing the compare function on all received messages. The proof can be found in the appendix.

2) Parallelization: Our simulation approach generates a control plane solution for each destination prefix. We can parallelize the phase of simulation for all destination prefixes by running multiple instances of simulation on separate machines. In addition, our verification approach does not store the computation outcomes for all possible failure scenarios and finds all violations of the property to be verified. On the contrary, it searches for only one failure scenario that violates the given property. Therefore, we can parallelize the property verification phase and terminate verification if any one node running the verification instance finds a violation.

3) Slicing: The intradomain routing paths are always irrelevant to the interdomain routing configurations. Therefore, we can use slicing optimizations by removing variables and constraints in the symbolic simulation process that cannot affect the final outcome of the intradomain endpoints. Similarly, the variables that never influence the routing decision process can be removed. For example, if BGP routers never set a local preference, then the local preference of the route attribute will never affect the decision.

V. Discussion

QNV is general and flexible, but it does have limitations. Like the methods demonstrated in [1], [8], [9], [10], [11], and [12], it describes stable solutions to which the control plane will converge. This choice makes it unable to support verifying quantitative policies about the transient states of a network. Unlike methods that try to monitor or detect transient performance violation [19], [20], [21], [22], [23], [24], QNV is designed to detect configuration vulnerabilities that could lead to serious performance violations, before the configurations are deployed. In addition, most operational diagnosis studies have used machine learning or statistical analysis for diagnosis with log messages or flow metrics; the methods developed therein perform learning on the normal system behaviors and used the learned models for diagnosis. However, these methods are ineffective at diagnosing rare anomalies or occasional glitches.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

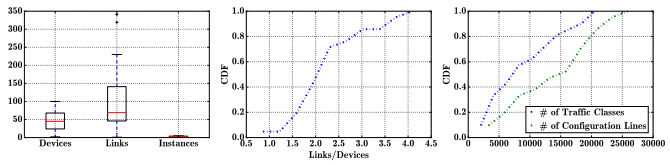

Fig. 7. Scale of networks evaluated in this study.

In future work, we will attempt to apply the knowledge from the symbolic simulation to help identify problems happening in the operation.

VI. Evaluation

QNV is implemented based on the configuration parser of Batfish [8]. The parser translates vendor-specific configurations (e.g., JunOS, Cisco IOS, Arista) into a vendor-agnostic configuration format. The symbolic simulation algorithm and property verification algorithm in Section IV are implemented in Java.

We evaluate QNV by using it to verify the policies described in Section IV-C on both real and synthetic network configurations. In particular, we are interested in measuring (i) the ability of QNV to find bugs in real configurations that could trigger quantitative property violation in the event of failures; (ii) the impact of the optimization strategies described in Section IV-D on the performance of QNV; and (iii) the scalability of QNV for verifying quantitative policies on large networks. All experiments were conducted on a server running the Ubuntu 10.12 operating system and equipped with an 8-core, 2.4-GHz Intel i7 processor.

A. Verification Efficiency

We demonstrate the ability of QNV to find bugs in real configurations. For this, we study a wide area network (WAN) and 20 datacenter networks operated by a large internet company. These networks use a combination of ISIS, OSPF, BGP, static routing constructs, ACLs, equal-cost multipath (ECMP), and route redistribution. Fig. 7 shows the scale of these networks. These networks have between 2 and 100 routers connected using between one and several hundreds of 100 Gbps links. The lines of configurations in these networks range from 200 to 25K each, and the traffic classes in each network range from 10 to 20K. The latencies of links, which are measured and offered by network managers, range from 18 μs to 300 ms. For each network, we generate 10 random traffic matrices using the gravity model [25]. As a result of the traffic matrices for each network make the maximum link utilization of the network ranges from 60% to 80%.

We evaluate the ability of QNV to detect link overloads for these networks with k link failure scenarios. We observe that the WAN experiences link overload for 6 traffic matrices when $k = 1$. When $k = 2$, there are 4 traffic matrices that cause link overload violations to occur in the WAN. When $k = 3$, there are 2 traffic matrices that cause the WAN to experience link overload. As k increases, the number of traffic matrices that cause the WAN to experience link overload decreases. This is as expected because the increased link failures disconnect more traffic classes, and thus the utilization of links does not exceed capacities. By comparison, Table V shows the results on link-load violation for the data-center networks. We observe that link overload violation occurs for 9 of the 20 data-center networks at 1 link failure. Moreover, 4 networks exhibit link overload violations for more than 5 out of 10 traffic matrices. When $k = 2$, six of the 20 datacenter networks experience link overload, and 3 networks exhibit link overload violations for more than 5 out of 10 traffic matrices. We observe that link overload violations in the data-center networks do not increase with the number of link failures. These results show that real WAN and data-center networks could suffer from link overload in the event of failures. These violations revealed optimization opportunities. Link being overloaded under failures could have been avoided using better routing and traffic engineering mechanisms.

TABLE V

Data-Center Networks That Experience Link Overload for One or More TMs (≥0%) and More Than 5 of 10 TMs (≥50%)

Class	(≥ 0%)	(≥ 50%)
$k = 1$	9	4
$k = 2$	6	3
$k = 3$	3	1

TABLE VI

Data-Center Networks That Experience Load Balancing Violation With One or More Link Pairs (≥0%) and More Than 5 of 10 Link Pairs (≥50%)

Class	(≥ 0%)	(≥ 50%)
$k = 1$	7	4
$k = 2$	8	3
$k = 3$	4	1

We next evaluate the ability of QNV to check the property of load balancing in the event of k link failures. For the WAN, we select 10 pair of links that are configured for load balancing. We observe that 6 of the 10 pairs of links that experience link balancing violation when $k = 1$. When $k = 2$ and $k = 3$, link balancing violations occur for 7 and 4 pairs of links, respectively. By comparison, Table VI shows the results on link balancing violations for the data-center networks. We observe that when $k = 1$, seven networks experience link balancing violations, and 5 networks experience load balancing violations for more than 5 out of 10 link pairs. When $k = 2$, eight of the 20 data-center networks experience link balancing violations, and 3 networks exhibit link overload violations for more than half of the selected link pairs. Meanwhile, when $k = 3$, four of the 20 data-center networks experience link balancing violation. We observe that, as k increases, the number of networks that experience link balancing violations does not increase. The result is as expected because the increased link failures disconnect more traffic classes.

We then examine the ability of QNV on verifying the latency property under k link failures for the WAN. When $k = 1$, 0.07% of the traffic classes in the WAN experience latency violation. When $k = 2$ and $k = 3$, only 0.23% and 0.19% of the traffic classes in the WAN experience latency

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

TABLE VII

PERCENTAGE OF VIOLATIONS FOR k LINK FAILURES FOUND IN THE WAN

Property	$k=1$	$k=2$	$k=3$
Latency	0.07%	0.38%	0.23%

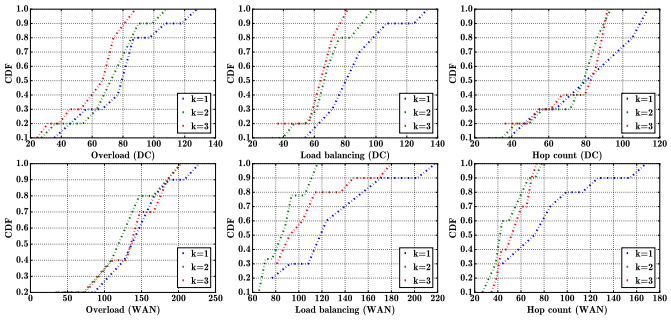

Fig. 8. Verification times for the WAN and the data-center networks.

violation. We can see that the latency violations for the traffic classes do not increase with the number of link failures. On the other hand, we do not examine the ability of QNV to verify the latency property on data-center networks. This is because, in the data-center networks, the time taken for one bit to travel from the sender to the receiver end of the links is not the performance bottleneck.

Finally, we evaluate the ability of QNV to check the hop-count property for these networks. For the WAN, we select 10 traffic classes that are marked as core services by the network manager; the traffic in these traffic classes never traverses a path longer than ten hops at 2 link failures. A violation of such a property likely indicates a configuration bug. We observe that 1 of the 10 traffic classes experience hop-count violations. For the 20 data-center networks, the network manager wants to ensure that the paths from all top-of-rack routers in a pod to a specific destination are never longer than 4 hops at 2 link failures. From the output of QNV, we observe that there are no data-center networks that experience hop-count violation at 2 link failures.

B. Optimization Effectiveness

We now evaluate how the optimizations presented in Section IV-D improve the performance of QNV on link overload verification.

Table VIII shows the speedups obtained via incremental updating (speedup = time without incremental updating/time with incremental updating) for the WAN and the data-center networks. When verification is performed on the WAN with $k = 2$, the speedup obtained via incremental updating is 1.3. On the other hand, for the data-center networks, the speedups for the DC4 and DC20 networks are 1.2 and 1.9, respectively. We observe that, for the bigger data-center networks, the speedup in verification is much larger.

We evaluate the speedup achieved by parallelizing the verification (speedup = time without parallelization/time with parallelization). In the parallel scenario, we run verification on 4 nodes. As shown in Table VIII, when verification is performed on the WAN with $k = 2$, the speedup obtained via parallelization is 1.7. On the other hand, for the data-center networks, the speedups for the DC4 and DC20 networks are 1.2 and 7.1, respectively. We observe that the parallelization in the bigger network improves the tractability of verification more significantly, and that the parallelization makes verification of QNV more tractable than that obtained via incremental updating.

TABLE VIII

SPEEDUPS DUE TO OPTIMIZATIONS FOR $k = 2$ FAILURES

Optimizations	WAN	DC4	DC8	DC12	DC16	DC20
Incremental updating	1.3	1.2	1.4	1.4	1.5	1.9
Parallelization	1.7	1.2	1.6	2.1	4.8	7.1
Slicing	2.2	1.3	1.4	1.3	1.3	1.4

Finally, we evaluate the speedup obtained via slicing (speedup = time without slicing/time with slicing). When verification is performed on the WAN with $k = 2$, the speedup obtained via slicing is 2.2. On the other hand, for the data-center networks, the speedups for the DC4 and DC20 networks are 1.3 and 1.4, respectively. We observe that the slicing in the WAN nimproves the tractability of verification, and that the speedup achieved via slicing does not increase with the size of the data-center networks. This is expected because the slicing only removes variables that do not affect the final outcome of the intradomain endpoints.

C. Comparing QNV With Existing Tools

To test the scalability of QNV on large networks, we use a collection of synthesized, but functional, configurations for data-center networks of increasing size [26]. There are three types of synthetic network topologies: Fattree, Ring, and Full-mesh. Each of these networks uses eBGP to perform shortest-path routing in conjunction with destination-based prefix filters to each destination. We generate 10 random traffic matrices using gravity model, as a result of the traffic matrices for each network, the maximum link utilization ranges from 60% to 80%. The value of the link latency does not affect the performance of QNV when the latency property is verified; therefore, the link latency in each network is assigned with a random value from 0.001 us to 0.009 us. For each network, we use QNV to verify the quantitative policies.

Batfish [8] is designed to verify Boolean policies such as reachability, multipath consistency, and destination consistency. One may argue that we can use Batfish to generate the complete data plane for every possible failure scenario and then use these data planes to verify the quantitative policies. To put our performance results into perspective, we run Batfish on the same machine and compare the cost of QNV with that Batfish. The time required by Batfish to verify each quantitative property across a set of link failure scenarios is a function of (i) the time required to generate the data plane, (ii) the number of scenarios, and (iii) the time to check the quantitative property with the data planes (time of Batfish = the time required to generate the data plane $\times$ the number of scenarios + the time to check the quantitative property with the data planes).

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

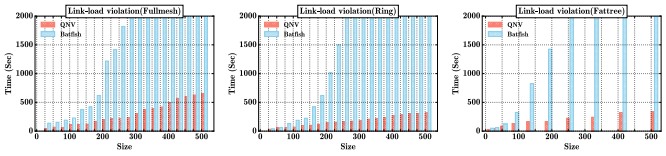

Fig. 9. Amounts of time to verify the link-load violation property for synthetic networks.

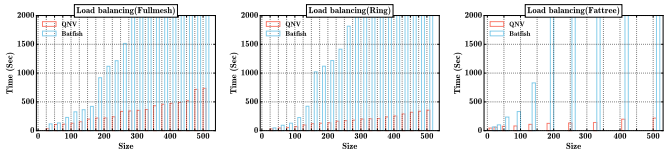

Fig. 10. Amounts of time to verify the load balancing property for synthetic networks.

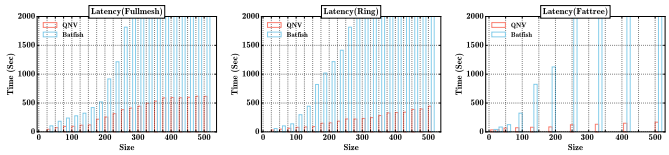

Fig. 11. Amounts of time to verify the latency property for synthetic networks.

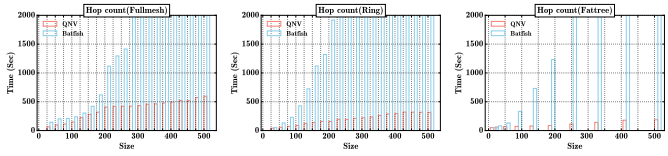

Fig. 12. Amounts of time to verify the hop-count violation property for synthetic networks.

We sort the networks by the number of routers and observe that verification time for these quantitative policies increases with the size of the network. As shown in Fig. 9, Fig. 10, Fig. 11, and Fig. 12, for the Full-mesh topologies with 2 link failures, QNV requires approximately 42 ms to 640 ms to verify the link-load violation property approximately 34 ms to 735 ms to verify the load balancing property, approximately 45 ms to 625 ms to verify the latency property, and approximately 65 ms to 595 ms to verify the hop-count property. For comparison, we also look at the performance of Batfish on the Full-mesh topologies; we observe that the time required by Batfish is approximately two orders of magnitude larger than the time required by QNV.

For the Ring topologies, QNV requires approximately 30 ms to 445 ms to verify these quantitative properties. We observe that the performance of QNV on Ring topologies is better than that on Full-mesh topologies. For the networks with an equal number of routers, verification for Ring topologies is less expensive than that the verification for Full-mesh topologies.

Finally, we evaluate the performance of Batfish and QNV on the Fattree topologies. As shown in Fig. 9, we observe that the quantitative policies can be checked to be approximately 20 ms to 340 ms. By contrast, the verification time of Batfish increases in an exponential way, making it impractical.

VII. Related Work

A. Control Plane Verification

Our work complements control plane verification tools, which focus on Boolean policies. Batfish [8] is the first control plane verification tool, which simulates routing protocols and analyzes the resulting data plane with a data plane verification tool. ARC [9] models the control plane as abstract weighted graphs, and with which it verifies the Boolean policies. ERA [10] abstracts the routing messages as the routing record and models the control plane as transfer functions of the routing records. Minesweeper [11] models the control plane as a satisfying assignment to an SMT formula. Plankton [12] uses model checking to explore control plane protocol behavior. Our work, which can verify the quantitative policies, is complementary to these efforts. The work closest to ours is ARC with quantities (QARC) [1]; it models the control plane as graphs and verifies whether a control plane may cause link-load violations in the event of specific failures. However, QARC cannot support verification in the event of an arbitrary number of failures. It cannot model either iBGP or certain features such as BGP local preference, which makes it impractical to be used.

B. Data Plane Verification

Recent efforts in network verification have also focused on network data planes [27], [28], [29], [30], [31], [32], [33]. Some of these verification tools [32], [33] check if a network property (e.g, reachability) would be satisfied on network data plane in a static way. For example, Anteater [27] is the first data plane verification tool, which translates Boolean network policies into satisfiability (SAT) problems, and checks them against forwarding information bases (FIBs) using an SAT solver. Header space analysis (HSA) [32] uses a geometric model of packet processing to verify a set of network policies. Other tools [34], [35] verify these policies in a real-time way by checking the changed FIBs. For example, VeriFlow [34] checks the behavior of the packet equivalence classes affected by the network update. Netplumber [35] verifies the policies by incrementally updating the transfer functions. The result from such verification, however, is valid only for the specific data plane. They cannot detect configuration vulnerabilities before the network configurations are deployed.

C. Probabilistic Network Verification

ProbNetKat [36] extends NetKat [37] to model probabilistic routing behavior and verifies network policies under a link failure probability distribution. However, it does not model the states at routers and cannot capture the behavior of control plane protocols. NetDice [38] supports probabilistic network verification by pruning the failure space for BGP and common interior gateway protocols (IGPs). It outputs a set of links whose failure is provably guaranteed not to change whether the policies hold. However, unlike, it cannot check whether the policies hold in the event of an arbitrary number of failures. Thus, our work is complementary to these efforts.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

VIII. Conclusion

Because networks are constantly changing (e.g., malicious route announcements from neighbors, failures caused by external attacks), network managers want the ability to analyze quantitative policies (e.g., link-load violation, load balancing, latency) across many possible changes. In contrast to prior work, which focuses on Boolean policies (e.g., reachability, forwarding loops), QNV supports the verification of quantitative policies in the event of an arbitrary number of failures. We have proposed a new symbolic simulation model that supports model packet forwarding behavior in the event of arbitrary failures and developed efficient algorithms to verify quantitative policies based on a new notion of the logical formula matrix. We use QNV on a collection of real and synthetic configurations, showing that it is effective at quantitative verification and can scale to large networks.

Appendix
Proof of Theorems

Theorem 1: Suppose node v currently has the route $\mathcal{F}(v) = \texttt{sort}\ ([r_1, r_2, r_3, \cdots r_i, \cdots r_k], \oplus)$ learned from neighbors; neighbor u now sends a new route $\bar{r_j}$. Then, we have $\texttt{sort}([r_1, r_2, r_3, \cdots r_i, \cdots r_k, \bar{r_j}], \oplus) = \texttt{sort}([\mathcal{F}(v), \bar{r_j}], \oplus)$.

Proof:

Suppose that $\mathcal{F}(v) = [r_1, r_2, r_3, \cdots r_i, \cdots r_k]$, $\bar{r_j} \oplus r_i = r_i$ and $\bar{r_j} \oplus r_{i+1} = \bar{r_j}$. Then, we have $\texttt{sort}([r_1, r_2, r_3, \cdots r_i, \cdots r_k, \bar{r_j}], \oplus) = [r_1, r_2, r_3, \cdots r_i, \bar{r_j}, r_{i+1}, \cdots r_k]$. Then, because $\oplus$ is commutative and associative, we determine that $\oplus$ has transitive property. $r_{i+1} \oplus \bar{r_j} = \bar{r_j}$, $r_{i+1} \oplus r_{i+2} \oplus \cdots r_k = r_{i+1}$, then we have $\bar{r_j} \oplus r_{i+1} \oplus r_{i+2} \oplus \cdots r_k = \bar{r_j}$. $\texttt{sort}([r_{i+1}, r_{i+2}, r_{i+3}, \cdots, r_k] \oplus \bar{r_j}) = [\bar{r_j}, r_{i+1}, \cdots r_k]$. Because $r_1 \oplus \bar{r_j} = r_1, \ldots, r_i \oplus \bar{r_j} = r_i$, then we have $\texttt{sort}([r_1, r_2, r_3, \cdots, r_k] \oplus \bar{r_j}) = [r_1, \cdots r_i, \bar{r_j}]$. Then, we have $\texttt{sort}([\mathcal{F}(v), \bar{r_j}], \oplus) = \texttt{sort}([[r_1, r_2, r_3, \cdots r_i, \cdots r_k], \bar{r_j}], \oplus) = \texttt{sort}([r_1, r_2, r_3, \cdots, r_i, \cdots, r_k] \oplus \bar{r_j}) = [r_1, r_2, r_3, \cdots r_i, \bar{r_j}, r_{i+1}, \cdots r_k]$. That is, $\texttt{sort}([r_1, r_2, r_3, \cdots r_i, \cdots r_k, \bar{r_j}], \oplus) = \texttt{sort}([\mathcal{F}(v), \bar{r_j}], \oplus)$. □

Lemma 1: $\oplus$ is commutative.

Lemma 2: $\oplus$ is associative.

Theorem 2: For the traffic tc, let r_i be the matched entry in the data plane matrix, and bdd be a BDD extracted from the condition of r_i. Assign weight 0/1 to solid/dashed edges of bdd. If ShortestPath(bdd, 0) $- 1 \geq k$ and LongestPath(bdd, 1) $\leq n$, we determine that the packets in tc sent from source can reach the destination in n hops in the event of k link failures, where ShortestPath(bdd, 0) is the length of the shortest path from the root node to terminal node 0, and LongestPath(bdd, 1) is the length of the longest path from the root node to terminal node 1.

Proof: Let Reach(tc) be the property that the packets in tc sent from the source are able to reach the destination. (i) Suppose ShortestPath(bdd, 0) is S. We prove the link failure tolerance F of the property Reach(tc) is $S-1$ by showing (1) $F \geq S-1$, and (2) $F < S$. (1) If $F < S-1$, we determine that the reachability does not hold when $S-1$ links are down and all other $N - S + 1$ links are up, where N is the total number of links in the network. In other words, a path to False has at most $S-1$ dashed edges, because the $N - S + 1$ links either correspond to solid edges or do not appear on the path. This contradicts the fact that the shortest path to False is S. (2) Because the shortest path to False is S, then there exists a case where S links are down, such that Reach(tc) does not hold. That is, we have the failure tolerance $F < S$. Then, we determine that if $F \geq k$, the packets in tc sent from the source can reach the destination under k link failures. (ii) Suppose LongestPath(bdd, 1) is L and the max hop count of the property Reach(tc) is T. We prove $L \geq T$. Suppose $L < T$; then, there exists a condition where T links are up and all other links are down, such that Reach(tc) holds. This contradicts the fact that the longest path to True is L. That is, if $T \geq n$, the packets in tc sent from the source can reach the destination in n hops. □

Theorem 3: The route attributes in DP are partitioned into failure scenario equivalence classes FL $= (R^0, R^1, \ldots, R^m)$ based on the conditions of the route attributes. If $\forall R \in$ FL. $\sum_{r_i \in R} \texttt{getLoad}(r_i.\text{prefix}, r_i.\text{length}) \leq \tau$, the load on link e does not exceed the threshold τ, where $\texttt{getLoad}(r_i.\text{prefix}, r_i.\text{length})$ is the traffic volume corresponding to the route attribute r_i.

Proof: Suppose there exists a set of failure scenarios R^j, where $R^j \notin$ FL, $\sum_{r_i \in R^j} \texttt{getLoad}(r_i.\text{prefix}, r_i.\text{length}) > \tau$, and the load on link e exceeds the threshold τ. We know that $\forall r_1 \in R^j$, $r_2 \in R^j$, r_1.cond $\wedge r_2$.cond $\neq$ False, getFalseValue(r_1.cond,r_2.cond)$< k$, and getLinkState(r_1.cond,r_2.cond,e)= True, where getFalseValue is the number of False variables in r_1.cond and r_2.cond, and getLinkState is the state of the link e in r_1.cond and r_2.cond. This contradicts the fact that FL consists of the failure scenario equivalence classes, where the intersection of the conditions of the route attributes in the same failure scenario equivalence class does not equal False. □

References

[1] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proc. 41st ACM SIGPLAN Int. Conf. Program. Lang. Design Implement. (PLDI)*, A. F. Donaldson and E. Torlak, Eds. London, U.K.: Association for Computing Machinery, 2020, pp. 974–988.

[2] *A Wide Range of Routers are Under Attack by New, Unusually Sophisticated Malware*. Accessed: Jun. 2022. [Online]. Available: https://arstechnica.com/information-technology/2022/06/a-wide-range-of-routers-are-under-attack-by-new-unusually-sophisticated-malware/

[3] , "Network misconfiguration risks and how to avoid them," Accessed: 2018. https://continuitycentral.com/index.php/news/technology/4990-common-network-misconfiguration-risks-and-how-to-avoid-them.

[4] *Misconfiguration Attacks: 5 Real-Life Attacks and Lessons Learned*. Accessed: 2020. [Online]. Available: https://brightsec.com/blog/misconfiguration-attacks/

[5] *Snowden: The NSA Planted Backdoors in Cisco Products*. Accessed: 2019. [Online]. Available: http://www.infoworld.com/article/2608141/internet-privacy/snowden–the-nsa-planted-backdoors-in-cisco-products.html

[6] *Facebook Blames Global Outage on Configuration Error*. Accessed: 2021. [Online]. Available: https://www.cnbc.com/2021/10/05/facebook-says-sorry-for-mass-outage-and-reveals-why-it-happened.html/.

[7] *Cloudflare Outage*. Accessed: 2020. [Online]. Available: https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/

[8] A. Fogel et al., "A general approach to network configuration analysis," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement.*, New York, NY, USA, 2015, pp. 469–483.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

[9] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Aug. 2016, pp. 300–313.

[10] S. K. Fayaz et al., "Efficient network reachability analysis using a succinct control plane representation," in *Proc. USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, New York, NY, USA, 2016, pp. 217–232.

[11] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, New York, NY, USA, Aug. 2017, pp. 155–168.

[12] S. Prabhu, K. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, R. Bhagwan and G. Porter, Eds. Berkeley, CA, USA: USENIX Association, 2020, pp. 953–967.

[13] P. Zhang, D. Wang, and A. Gember-Jacobson, "Symbolic router execution," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2022, pp. 336–349.

[14] G. Juniwal, S. A. Seshia, and G. Varghese. (2015). *Quantitative Network Analysis*. [Online]. Available: https://cseweb.ucsd.edu//~varghese/qna.pdf/

[15] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, Dec. 2009.

[16] F. Ye et al., "Accuracy, scalability, coverage: A practical configuration verifier on a global WAN," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, New York, NY, USA, Jul. 2020, pp. 599–614.

[17] *Cisco IOS Master Command List, All Releases*. Accessed: Jun. 2014. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/mcl/allreleasemcl/allbook.html

[18] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[19] L. Huang and T. Zhu, "Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Seattle, WA, USA, C. Curino, G. Koutrika, and R. Netravali, Eds. New York, NY, USA: Association for Computing Machinery, 2021, pp. 76–91.

[20] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, Belgrade, Serbia, G. Alonso, R. Bianchini, and M. Vukolic, Eds. New York, NY, USA: Association for Computing Machinery, 2017, pp. 64–79.

[21] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, p. 11, 2017.

[22] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proc. ACM SIGCOMM Conf.*, Chicago, IL, USA, F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds. New York, NY, USA: Association for Computing Machinery, 2014, pp. 527–538.

[23] Y. Chang, S. G. Rao, and M. Tawarmalani, "Robust validation of network designs under uncertain demands and failures," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, A. Akella and J. Howell, Eds. Berkeley, CA, USA: USENIX Association, 2017, pp. 347–362.

[24] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *Proc. IEEE INFOCOM*, Washington, DC, USA, vol. 2, Mar. 2000, pp. 519–528.

[25] M. Roughan, A. G. Greenberg, C. R. Kalmanek, M. P. Rumsewicz, J. Yates, and Y. Zhang, "Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning," in *Proc. 2nd ACM SIGCOMM Internet Meas. Workshop (IMW)*, Marseille, France, New York, NY, USA: Association for Computing Machinery, 2002, pp. 91–92.

[26] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, New York, NY, USA, Aug. 2018, pp. 476–489.

[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Aug. 2011, pp. 290–301.

[28] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2016, pp. 314–327.

[29] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Berkeley, CA, USA, 2015, pp. 499–512.

[30] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *Proc. 14th USENIX Conf. Netw. Syst. Design Implement.*, Berkeley, CA, USA, pp. 735–749.

[31] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, New York, NY, USA, Jan. 2016, pp. 69–83.

[32] P. Kazemian, G. Varghese, and N. Mckeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, Berkeley, CA, USA, 2012, p. 9.

[33] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proc. 14th USENIX Symp. Networked Syst. Design Implement.*, Boston, MA, USA, Mar. 2017, pp. 699–718.

[34] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *ACM SIGCOMM Comput.*, vol. 42, no. 4, pp. 467–472, Sep. 2012.

[35] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Lombard, IL, USA, N. Feamster and J. C. Mogul, Eds. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–111.

[36] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *Proc. Program. Lang. Syst.-25th Eur. Symp. Program., ESOP, Held Eur. Joint Conf. Theory Pract. Softw. (ETAPS)* (Lecture Notes in Computer Science), Eindhoven, The Netherlands, vol. 9632, P. Thiemann, Ed. New York, NY, USA: Springer, 2016, pp. 282–309.

[37] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for netkat," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, Mumbai, India, S. K. Rajamani and D. Walker, Eds. New York, NY, USA: Association for Computing Machinery, 2015, pp. 343–355.

[38] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. T. Vechev, "Probabilistic verification of network configurations," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Communication. (SIGCOMM)*, Schulzrinne and V. Misra, Eds. New York, NY, USA: ACM, 2020, pp. 750–764.

Yahui Li received the B.S. degree from the School of Software Engineering, Jilin University, Changchun, China, and the Ph.D. degree from Tsinghua University, Beijing, China. She is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University. Her research interests include computer network systems and network security.

Han Zhang (Member, IEEE) received the B.S. degree in computer science and technology from Jilin University, Changchun, China, and the Ph.D. degree from Tsinghua University, Beijing, China. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include computer network systems and network security.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.

Jilong Wang (Senior Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2000. He is currently a Professor with Tsinghua University. His research interests include network measurement, location-oriented networks, SDN systems, and network security.

Xingang Shi (Member, IEEE) received the B.S. degree from Tsinghua University and the Ph.D. degree from The Chinese University of Hong Kong. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network measurement and routing protocols.

Xia Yin received the B.E., M.E., and Ph.D. degrees in computer science from Tsinghua University, China, in 1995, 1997, and 2000, respectively. She is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. Her research interests include future internet architectures, formal methods, protocol testing, and large-scale internet routing.

Zhiliang Wang received the B.E., M.E., and Ph.D. degrees in computer science from Tsinghua University, China, in 2001, 2003, and 2006, respectively. Currently, he is an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include formal methods and protocol testing, next-generation internet, and network measurement.

Jiankun Hu (Senior Member, IEEE) received the Ph.D. degree in control engineering from Harbin Institute of Technology, China, in 1993, and the master's degree in computer science and software engineering from Monash University, Australia, in 2000. He was a Research Fellow with Delft University of Technology, The Netherlands, from 1997 to 1998, and Melbourne University, Australia, from 1998 to 1999. He was with Ruhr University, Bochum, Germany. He is currently a Full Professor and the Research Director of the Cyber Security Laboratory, School of Engineering and Information Technology, University of New South Wales, Australia. His main research interests include cyber security and biometrics security, where he has published many papers in high-quality conferences and journals, including IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE. He received the prestigious German Alexander von Humboldt Fellowship from Ruhr University from 1995 to 1996. He has received seven Australian Research Council (ARC) Grants and serves at the prestigious Panel of Mathematics, Information and Computing Sciences and the ARC Excellence in Research for Australia Evaluation Committee. He was the Security Symposium Chair of the IEEE ICC and the IEEE Globecom. He has served on the editorial board for seven international journals, including IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY.

Congcong Miao received the B.E. degree from Beijing University of Posts and Telecommunications and the Ph.D. degree from Tsinghua University. His research interests include network measurement and management, network data analysis with machine learning method, and spatio-temporal data mining.

Jianping Wu (Fellow, IEEE) received the B.S., M.S., and Ph.D. degrees from Tsinghua University. He is currently a Full Professor and the Director of the Network Research Center and the Ph.D. Supervisor with the Department of Computer Science, Tsinghua University. Since 1994, he has been in charge of China Education and Research Network (CERNET). His research interests include next-generation internet, IPv6 deployment and technologies, and internet protocol design and engineering. He is a member of the Information Advisory Committee, Office of National Information Infrastructure, a Secretariat of State Council of China, and a Vice President of the Internet Society of China (ISC).

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:51:03 UTC from IEEE Xplore. Restrictions apply.