A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures

Ruihan Li[1,2], Yifei Yuan[2], Fangdan Ye[2], Mengqi Liu[2], Ruizhen Yang[2], Yang Yu[2], Tianchen Guo[2], Qing Ma[2], Xianlong Zeng[2], Chenren Xu[1], Dennis Cai[2], Ennan Zhai[2]
[1]Peking University [2]Alibaba Cloud

ABSTRACT

This paper presents YU, the first verification system for checking traffic load properties under *arbitrary* failure scenarios that can scale to production Wide Area Networks (WANs). Building a practical YU requires us to address two challenges in terms of *generality* and *efficiency*. The state-of-the-art efforts either assume shortest-path-based forwarding (*e.g.*, QARC) or only target single-failure reasoning (*e.g.*, Jingubang). As a result, the former inherently cannot generalize to widely used protocols (*e.g.*, SR and iBGP) that are beyond shortest-path forwarding, while the latter cannot efficiently handle arbitrary failure scenarios. For the generality challenge, we propose an approach inspired by symbolic execution, called symbolic traffic execution, to model the forwarding behavior of a range of practically deployed protocols (*e.g.*, eBGP, iBGP, iGP, and SR) under failure scenarios. For the efficiency challenge, we propose diverse equivalence classification techniques (*i.e.*, k-failure-equivalence and link-local-equivalence reduction) to reduce the symbolic traffic execution overhead caused by both the large size of the production WAN and the huge number of traffic flows traversing it. YU has been used in the daily verification of our WAN for several months and has successfully identified potential failure scenarios that would lead to traffic load violations.

CCS CONCEPTS

• **Networks** → **Network reliability**; **Network manageability**; • **Theory of computation** → **Automated reasoning**.

KEYWORDS

Network Verification; Traffic Load Property; Network Configurations; Symbolic Execution

ACM Reference Format:
Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. 2024. A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672246

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672246

1 INTRODUCTION

Alibaba Cloud's global wide area network (WAN) infrastructure interconnects tens of data centers through ~ 1000 routers. These routers are operated in a distributed setting, running a diverse set of protocols including BGP [2], IS-IS [3], and segment routing (SR) [19]. This WAN supports core services such as cloud computing and e-commerce, serving more than one billion customers globally.

As this WAN carries and forwards a substantial amount of service traffic every day, failures in links or routers may cause abnormal traffic load changes on links (*e.g.*, due to unexpected changes in the traffic's forwarding paths), resulting in severe service outages. For example, a link failure in a SR tunnel may lead to traffic unexpectedly falling back to tunnels with bottleneck links, causing the links to be overloaded, and thus downgrading the QoS of the hosted service (see §6 for more real-world examples). According to our recent five-year internal network outage records, more than 90% of the outages are caused by traffic load violations, with a majority of them caused by failures in links and routers. Therefore, *proactively* verifying whether the network meets *traffic load properties* (TLP) (*e.g.*, no overloaded links) under arbitrary failures (up to a given degree k) is vital to the availability and reliability of the network. We refer to this problem as the *k-failure TLP verification* problem.

Why prior work does not help? State-of-the-art network verification systems have been focused on checking reachability properties (*e.g.*, if routes/packets from a router C can reach another router D) in terms of control plane [4, 8, 17, 20, 23, 24, 33, 46, 47, 50, 55–57, 59] and data plane [9, 29, 31, 32, 34, 35, 43, 45, 51, 54]; however, they are not designed to verify TLPs in the network.

In recent years, we have seen some first-step exploration on the k-failure TLP verification problem. QARC [52] proposed a verification technique to detect overloaded links when at most k links can fail. QARC assumes that traffic forwarding always follows the shortest paths, and thus models the network control plane as a weighted graph. This modeling allows the k-failure TLP verification problem to be encoded as an integer linear programming problem, which can be solved effectively by modern solvers. However, QARC's shortest-path-based modeling is fundamentally limited in modeling a number of key features used in production networks, such as SR, iBGP, and BGP local preference, under which traffic forwarding is beyond the shortest paths. Moreover, QARC is unlikely to scale to production WANs with thousands of links (see §7.2). Another recent work Jingubang [39] proposed a general verification algorithm, supporting a wide range of features (*e.g.*, iBGP/eBGP, SR), to check TLPs for production networks. While Jingubang proposed an efficient incremental algorithm to verify TLPs under *a specific* failure scenario, it is hard to extend the algorithm to efficiently check TLPs under *arbitrary* k-failure scenarios, which can be prohibitively many. Thus, the fundamental research question that, how

to build a verification system for k-failure TLP verification at the production scale while supporting practically deployed features, still remains open.

1.1 Our Approach: YU

This paper presents YU, the first verification system for k-failure TLP verification targeting production networks.[1]

Generality. In order to support various network features (beyond shortest-path forwarding), YU must model the forwarding behavior of each flow on every router at the foundation level, including basic primitives such as route selection, (weighted) equal-cost multipath (ECMP) forwarding, and route iteration, for both IP and SR forwarding. Meanwhile, YU must be able to reason about all possible forwarding behaviors under arbitrary k-failure scenarios.

Inspired by the principle of symbolic execution on computer programs [36], YU adopts a general *symbolic traffic execution* approach, by viewing the forwarding process of a given flow in the network as a program and the failure state of links/routers as the input. First, YU conducts symbolic route simulation [57] for all used routing protocols in our network and generates guarded routing tables, where each route is guarded with a constraint encoding all the scenarios under which the route can present. Second, for each flow entering the network, YU simulates its forwarding process on each router, including route selection and ECMP, while recording and propagating the constraints encountered in each step. As a result, YU computes a symbolic traffic load for each link of the flow, encoding the mapping of each failure scenario to a concrete traffic load number. Finally, with the symbolic traffic load on each link, a TLP can be effectively verified by solving the constraint (§4).

Efficiency. With the general symbolic traffic execution approach, the key challenge then is how to improve its efficiency to the production network scale. First, given the large number of routers and links in the network, the number of failure scenarios can be prohibitively large. Thus, a naive encoding of constraints in the symbolic traffic execution can grow extremely fast. Second, the network forwards billions of flows. Thus, verifying TLPs for a link by trivially combining all flows' symbolic traffic load on it induces huge overhead.

To address the two efficiency challenges above, YU makes the following two contributions accordingly. First, to compactly encode the constraints, YU employs *multi-terminal binary decision diagrams* (MTBDDs) [6, 15, 22], which succinctly represents a constraint as a single-source-multiple-sink directed acyclic graph (DAG). We observe that the MTBDD can be greatly simplified under the k-failure constraint while maintaining the correctness of verification. For example, if a MTBDD encodes more than k failed links, then it can be pruned, since such a failure scenario is beyond the degree k. Motivated by this insight, we propose a novel MTBDD k-failure-equivalence reduction technique that effectively reduces the size of MTBDDs propagated in the symbolic traffic execution process.

Second, while different flows typically have different global forwarding behavior in the network, they may exhibit the same forwarding behavior on a specific link under all failure scenarios. As a result, the MTBDDs encoding the flows' symbolic traffic load on the link are equivalent to one another. This *link-local* flow-equivalence relation allows us to greatly reduce the number of flows to be checked on a link by replacing a large number of equivalent flows with one, thus improving the efficiency of verification for that link.

YU has been used in the daily verification of our WAN for several months. Our operators used YU to check crucial TLPs and successfully identified a number of failure scenarios under which those properties would be violated, leading to severe outages. Our performance evaluation based on our production networks shows that YU (1) only takes tens of minutes to verify k-failure TLPs for our global network with more than one thousand routers and (2) is orders of magnitude faster than alternative approaches.

Ethics. This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

Background. Alibaba Cloud's global WAN consists of $\sim$ 1000 routers and thousands of links, interconnecting tens of data centers and external ISP peers. This WAN runs a traditional distributed control plane, using protocols such as BGP (eBGP and iBGP), IS-IS, and SR. The mainstream SR used in this WAN is Segment Routing IPv6 (or SRv6). As of Jan 2024, this WAN has millions of IP prefixes and carries billions of flows.

The WAN is intended to tolerate some degree of failures (*e.g.*, link failures) such that no critical traffic load properties are violated, *e.g.*, no overloaded links and no unexpected traffic load dropping.

2.1 Motivating Example

Inspired by real configurations in our WAN, Figure 1 illustrates an example that motivates the need for a new verification system checking traffic load properties under arbitrary failure scenarios.

This example network consists of six routers (router A-F), connected with 100 Gbps links. Those routers are configured into three autonomous systems (AS 100-300) and run eBGP on the border to exchange routes. Routers in AS 300 runs iBGP with IS-IS as IGP, where the IGP cost is 10000 for all links and 5 for loopback interfaces. Router D is configured with a segment routing (SR) policy that forwards matched traffic (*i.e.*, DSCP=5) through paths D-E-F and D-C-F with weights 75 and 25 respectively. Green boxes show the BGP and IS-IS RIBs for selected routers; the yellow box shows the SR configuration of router D. The RIBs and configurations of other routers are omitted due to space limits.

The network forwards two flows f_1 and f_2 to the destination 100.0.0.0/24, with each flow carrying 20 Gbps and 80 Gbps traffic, respectively. To offer high availability and reliability, the network is intended to maintain the following two TLPs: (P1) the traffic load delivered to the destination should not drop significantly (*e.g.*, $<$ 70 Gbps) and (P2) no link is overloaded (*e.g.*, $\geq$ 95 Gbps).

Clearly, when no link fails, both P1 and P2 are satisfied, as shown by the traffic load on each link in Figure 1(a). Note that router B forwards f_2 to both C and D via ECMP; thus each one of the links B-C and B-D carries 40 Gbps traffic, respectively.[2] Based on the configured SR policy, router D forwards $75/(75+25) = 75\%$ of f_2's

[1] In ancient Chinese mythology, Yu the Great is famed for his efforts at flood control. Therefore, we named our network traffic load verification system YU.

[2] We assume that traffic is distributed equally across multiple equal-cost paths or proportionally according to the pre-configured weights. This assumption is reasonable given our WAN settings, as detailed in [39].

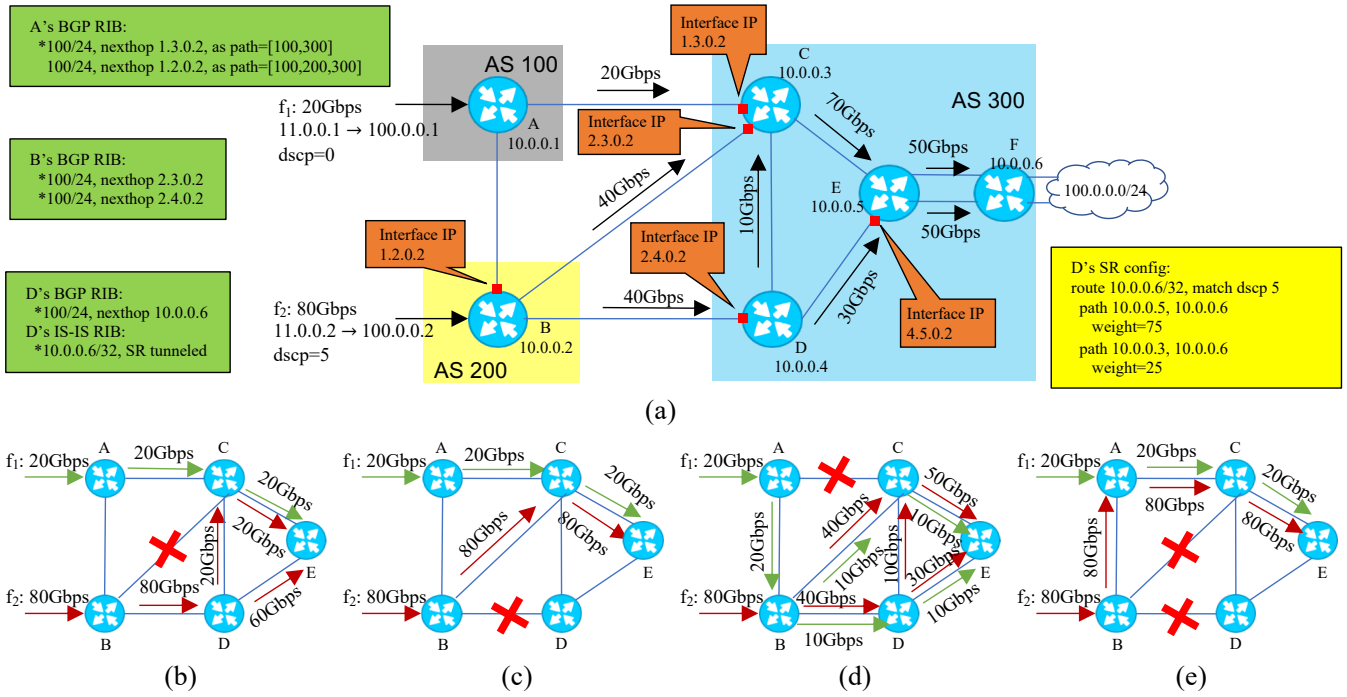


Figure 1: Motivating example: (a) shows the non-failure scenario. The network is configured with eBGP/iBGP, IS-IS, SR and forwards the traffic of two flows. The green boxes show the routing tables of router A, B, D. Selected routes are labeled with *. The yellow box shows router D's SR policy. Each link is labeled with the amount and direction of the traffic it forwards. (b)-(e) show how the two flows' traffic is forwarded on each link under specific failed links (labeled with red-cross markers), f_1's traffic is labeled in green and f_2's traffic is in red.

incoming traffic to E and $25/(75+25) = 25\%$ of that to C; thus, link D-E and D-C carry 30 Gbps and 10 Gbps traffic of f_2, respectively.

To ensure that the network can tolerate some degree of failures, network operators may want to check whether P1 and P2 still hold under arbitrary link failures. For example, when any single link can fail, it can be checked that P1 always holds. However, when link B-D fails in the network, all traffic of f_2 has to be forwarded to router C, which forwards all 100 Gbps traffic of both flows via link C-E (as shown in Figure 1(c)), causing the link to be overloaded. Therefore, while P1 is satisfied under all considered scenarios, P2 would be violated when link B-D failed, resulting in a severe outage.

Why existing approaches do not help? Route verifiers (*e.g.*, Minesweeper [8] and Hoyan [57]) mainly focus on checking route reachability properties (*e.g.*, whether a route advertised from A can reach B) and are not easily extensible to verify TLPs. On the other hand, recently proposed TLP verification systems, QARC and Jingubang, face fundamental difficulties in their *generality* and *efficiency*. QARC critically relies on the shortest-path-forwarding assumption, thus, it cannot be applied to the example network above (*e.g.*, f_2's forwarding path with SR is not the shortest). While Jingubang proposes a general model to support a wide range of features, it can only verify a single failure scenario at a time. Thus, to detect the risk in the example, Jingubang needs to enumerate and check all possible failure scenarios, which is inefficient.

2.2 Technical Challenges

Designing a general verification system checking TLPs under arbitrary failures faces significant challenges in its *efficiency*:

C1: Reasoning a large space of traffic forwarding behaviors under failures. A flow's forwarding behaviors vary significantly under different failure scenarios. In our motivating example, f_2 is forwarded by router B to both C and D via ECMP, with each link taking 50% of the flow's traffic load, when no failure happens. However, when one of the links B-C and B-D fails, f_2 is forwarded to the other link with 100% traffic load (Figure 1(b) and 1(c)); in the scenario that both B-C and B-D fail, f_2 is forwarded to A with 100% traffic load (Figure 1(e)). Such fluctuation further leads to cascading effects along the entire forwarding path across the network (*e.g.*, the far-away links C-E and D-E). As a result, the space of all possible forwarding behaviors can be as large as the entire failure scenario space, which is of size $O(n^k)$ (n denotes the size of the network, *e.g.*, the number of links). This makes the verification system hard to scale for production networks with thousands of links.

C2: Reasoning a large number of flows. As production networks employ a rich set of network features, flows' forwarding behaviors differ greatly from one another, especially under various failure scenarios. For example, as shown in Figure 1(b)-(e), the network-wide forwarding behaviors of f_1 and f_2 differ in each failure scenario. Thus, it is hard to reason traffic loads under failures by simply

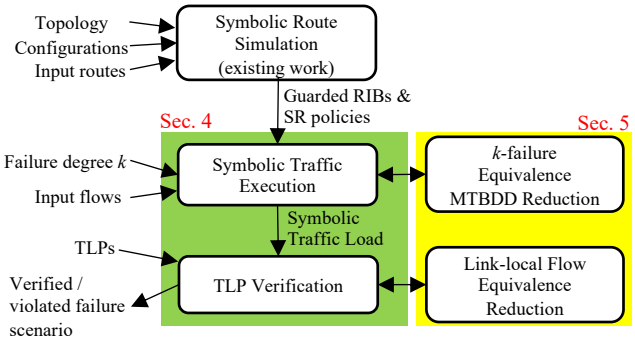


Figure 2: YU's high-level workflow.

treating a large number of flows as having identical effects. As a result, to verify TLPs for a link, a trivial approach may have to reason about traffic introduced by a large number of flows on the link, which is inefficient. Thus, the large number of flows in production networks remains a significant challenge.

3 OVERVIEW

In this section, we discuss YU's key insights to address the above challenges, and then describe its high-level workflow.

3.1 Key Insights

YU's design is driven by the following key insights.

I1: Symbolic traffic execution. To model the complex traffic behavior under various network features, one may have to simulate the forwarding behavior of each flow, similar to the approach in Jingubang [39]. However, given the large space of traffic forwarding behaviors under failure (C1) this *concrete* traffic simulation approach has to enumerate all failure scenarios and simulate the traffic load under each scenario, which is inefficient. YU adopts a *symbolic execution* approach (which we refer to as *symbolic traffic execution*), where each router/link's state (*i.e.*, alive or failed) is encoded as a boolean variable and the traffic load of a flow on each link is then represented symbolically as a function of those variables. The symbolic traffic execution allows YU to compute the traffic load in *all* failure scenarios in a single run of execution. It not only avoids running concrete simulations mulitple times (up to $O(n^k)$), but also offers opportunities to summarize similar forwarding behaviors among different flows across various failure scenarios (see below), thereby reducing redundant computations significantly.

I2: Compact symbolic representation and size reduction. A key challenge of the symbolic traffic execution approach, induced by the large space of traffic forwarding behavior (C1), is that the symbolic representation of traffic loads may grow fast and quickly exhaust system resources. Thus, such a representation must be *compact* throughout the entire symbolic execution. To this end, we investigate a variant of binary decision diagrams [12], called multi-terminal binary decision diagrams (MTBDDs), to compactly encode such representations. To avoid the exponential explosion of MTBDDs during symbolic traffic execution, we propose novel MTBDD size reduction techniques based on k-failure equivalence. By replacing exact encodings with k-failure equivalent ones, we can effectively reduce the size of a MTBDD while still ensuring verification correctness under scenarios with no more than k failures.

I3: Equivalence-relation reduction for efficient checking. With the MTBDD representation of traffic loads of each flow, checking a TLP on a link requires to add a large number of MTBDDs for all flows (C2). To address this challenge, we establish *link-local equivalence relations* among flows, allowing us to consider a significantly smaller number of flows when verifying TLPs regarding each link.

Table 1: Comparison with the state-of-the-art efforts.

	Generality				Efficiency
	eBGP	iBGP	IGP	SR	for k-failure TLP
QARC [52]	Y	N	Y	N	Low
Jingubang [39]	Y	Y	Y	Y	Low
YU	**Y**	**Y**	**Y**	**Y**	**High**

3.2 Workflow

Figure 2 shows the high-level workflow of YU. First, taking the (i) network topology, (ii) configurations of all routers, and (iii) all input routes as input, YU runs symbolic route simulation [57] to compute the *guarded RIB* and *guarded SR policies* for each router. In a guarded RIB, each route is guarded with a constraint, encoding all the scenarios under which the route can appear on the router.

Second, given the guarded RIB and SR policies of each router and all the input flows, YU runs symbolic traffic execution for each input flow. YU computes a MTBDD for each flow and each link as the symbolic representation of the traffic load, *i.e.*, the symbolic traffic load (STL) (§4). Meanwhile, YU actively applies size reduction techniques to reduce the MTBDD size to improve efficiency (§5.2).

Finally, YU checks the desired TLP on each link. Specially, YU aggregates the STLs of all flows on that link, and checks whether there exists a solution to the violation of the TLP for the aggregated STL. If that is the case, YU finds a failure scenario and link where the TLP is violated and our operators will manually analyze the failure scenario to locate the root cause (§6); otherwise, the TLP is satisfied in all failure scenarios with no more than k failures. To improve the efficiency of checking, YU identifies link-local equivalent flows (§5.3), which reduces the number of flows to be considered.

Traffic load properties of interest. Similar to Jingubang, we define a TLP as required ranges of traffic loads enforced on a list of specific links. Formally, a TLP is a set of $\{l_i : [v_1, v_2]\}$ pairs, where l_i is a link and $[v_1, v_2]$ specifies the range of required traffic loads.

Comparison with existing work. Revisiting the existing work discussed in §2, we highlight the difference of YU in Table 1. Only YU meets the generality goal of supporting a wide range of network features, and can efficiently check TLPs for production networks under arbitrary k failures.

4 SYMBOLIC TRAFFIC EXECUTION BASED k-FAILURE TLP VERIFICATION

Given a set of flows entering a network, YU symbolically executes the forwarding process of each flow's traffic in the network, viewing the state of each router/link as a symbolic variable. In this section,

Table 2: Summary of notations.

	Description	Range
$r_1 \prec r_2$	rule r_1 has higher preference than r_2	-
w_p	the weight of a SR path p	-
$l_1, l_2, ...$	network links	-
$x_1, x_2, ..., y$	the state of links	$\{0,1\}$
g_r (g_p)	guard of a rule r (a SR path p)	$\{0,1\}$
s_r^{ip}	whether r is selected for the destination IP ip	$\{0,1\}$
ω_l^f	symbolic traffic fraction (STF) of f on link l	$[0,1]$
ω_R^f	symbolic traffic fraction (STF) of f on router R	$[0,1]$
τ_l	symbolic traffic load (STL) of link l	$\mathbb{R}$

we elaborate on the design of the symbolic traffic execution algorithm, focusing on link failures for presentation purposes (router failures are supported similarly). First, we introduce the guarded RIBs and SR policies generated by symbolic route simulation. Then we describe how to represent traffic load on a link symbolically. Next, we describe the symbolic traffic execution framework and YU's symbolic encodings of forwarding behavior.

Notations and terminologies. We consider a flow as a (*intf*, *srcip*, *dstip*, *dscp*) tuple, indicating all the packets entering the network via the interface *intf* with the source IP *srcip*, destination IP *dstip*, and DSCP value *dscp*. We use V_f to denote the total traffic volume of the flow f. For SR forwarding, packets in a flow are forwarded in tunnels with label stacks indicating the configured list of routers in the SR path (YU also supports SR policies specifying outgoing interfaces; the details are omitted due to space limit). For simplicity, we use the list of routers in the SR path to denote its label stack, *e.g.*, [E, F] denotes the label stack of the first SR path configured on router D (Figure 1). To model the direction of a flow forwarded on a link, we model a network link with directions, and use the term incoming/outgoing links to refer to the links with corresponding forwarding directions. Table 2 summarizes commonly used notations in this paper.

4.1 Symbolic Route Simulation

YU employs the techniques proposed in [57] to run symbolic route simulation and generates the symbolic representation of RIBs (including BGP and IS-IS) and SR policies, which we refer to as *guarded RIBs* and *guarded SR policies*, respectively.

Guarded RIBs. A guarded RIB extends a concrete RIB with a symbolic constraint (called guard) for each route, encoding the scenarios where the route can present in the router's RIB. For the motivating example, the guarded (BGP) RIB of router A is shown in Figure 3. Here, for brevity of illustration, we only consider the state of three links, *i.e.*, A-C, B-C, and B-D, denoted as variable x_1, x_2, and x_3, respectively (*e.g.*, $x_1 = 1$ if A-C is alive, otherwise $x_1 = 0$).

In the guarded RIB, there are two rules, namely r_1 and r_2, corresponding to the two concrete routes in Figure 1. r_1 is guarded with constraint x_1, indicating that r_1 can appear in the RIB if and only if A-C is alive. Similarly, r_2 is guarded with $x_2 \vee x_3$, denoting r_2 can appear if and only if either B-C or B-D is alive. A guard only constrains the presence of a route without changing any attribute. Thus, the precedence of rules in a guarded RIB is determined in the normal fashion. For the example above, r_1 is preferred to r_2 (denoted $r_1 \prec r_2$), since r_1's AS path is shorter than that of r_2.

Figure 6 briefly demonstrates the symbolic route simulation process for the guarded RIB; we refer to §5 of [57] for the detailed algorithm. In the symbolic route simulation, a route advertisement message is also extended with a guard. For example, the advertisement message m_1 sent from router C to router A has the guard x_1, indicating that the message can be sent when link A-C is alive. As shown in step (1)-(3) in the figure, routers in AS 300 first sends messages m_1-m_3 to router A and B. After receiving m_2 and m_3, router B generates message m_4 and sends it to A, as shown in step (4). The guard of m_4 is the disjunction of that of m_2 and m_3 (namely, $x_2 \vee x_3$), since the routes in m_2 and m_3 have equal precedence. Finally, after receiving m_1 and m_4, router A can generate the guarded RIB.

Guarded SR policies. Similarly, in a guarded SR policy, each SR path is guarded with a constraint encoding the scenarios where the SR tunnel for the path can be established. For example, Figure 4 shows the guarded SR policy on router D. In order to establish the SR tunnel D-E-F, *i.e.*, p_1, the following two conditions must be met: (1) router D can reach router E via IS-IS in AS 300 and (2) router E can reach router F via IS-IS in AS 300. We denote the two conditions as $reach_{D,E}$ and $reach_{E,F}$. Thus, the guard of p_1 can be computed as $reach_{D,E} \wedge reach_{E,F}$. The guard of p_2 can be computed similarly.

As an example, we illustrate the computation of $reach_{D,E}$, by only considering the state of link C-E (denoted as variable y_1) and link D-E (denoted as variable y_2). First, we run symbolic router simulation to generate the guarded IS-IS RIB for all routers in AS 300 (the process is similar to the one shown in Figure 6). For the simple example, router D's guard IS-IS RIB has two routes to router E: (1) a preferred route with the next hop directly to E and guard y_2, and (2) a less preferred route with the next hop to C and guard y_1. As a result, $reach_{D,E}$ is computed as the disjunction of the two guards, namely $reach_{D,E} = y_1 \vee y_2$.

4.2 Symbolic Traffic Load and Fraction

By encoding each link's state as a symbolic variable, the traffic load on a link l in a set of scenarios can be represented as a symbolic formula τ_l, which we refer to as *symbolic traffic load* (STL). We also use τ_l^f to denote the STL on l but only considering f's traffic. We also represent the fraction of traffic of a flow f on a link l (a router R, resp.) as a symbolic formula ω_l^f (ω_R^f, resp.), which we refer to as *symbolic traffic fraction* (STF). Since $\tau_l^f = V_f * \omega_l^f$ where V_f is total traffic volume of f, we will use STF in the process of symbolic traffic execution discussed later, as it offers additional convenience (see §5.3). STLs and STFs are functions mapping $\{0,1\}^n$ to real numbers. They are known as *pseudo-boolean functions* [27]. We consider a symbolic constraint (*e.g.*, a guard) also as a pseudo-boolean function, for ease of operation.

Example. Consider the flow f_1 and link C-E in the scenarios shown in Figure 1(a)-(e). Similar to the previous example, we only consider the state of link A-C, B-C, and B-D (with x_1, x_2, and x_3 as the symbolic variables, respectively) for brevity. The STF of f_1 on link C-E is shown in Figure 5.

To explain, consider failure scenario (b), where only link B-C fails. The link state of this scenario can be represented as $x_1\overline{x_2}x_3$, where x_1 and x_3 denote that A-C and B-D are alive, and $\overline{x_2}$ denotes that B-C is failed, correctly encoding the failure scenario. Since link

	Prefix	Next Hop	AS Path	Guard
r_1:	100.0.0.0/24	1.3.0.2	[100,300]	x_1
r_2:	100.0.0.0/24	1.2.0.2	[100,200,300]	$x_2 \vee x_3$

Figure 3: Guarded RIB of router A.

	SR path	Weight	Guard
p_1:	[E, F]	75	$reach_{D,E} \wedge reach_{E,F}$
p_2:	[C, F]	25	$reach_{D,C} \wedge reach_{C,F}$

Figure 4: Guarded SR policy of router D.

scenario (a):	$1 * x_1x_2x_3$
scenario (b):	$1 * x_1\overline{x_2}x_3$
scenario (c):	$1 * x_1x_2\overline{x_3}$
scenario (d):	$0.5 * \overline{x_1}x_2x_3$
scenario (e):	$1 * x_1\overline{x_2}\overline{x_3}$

Figure 5: f_1's STF on C-E

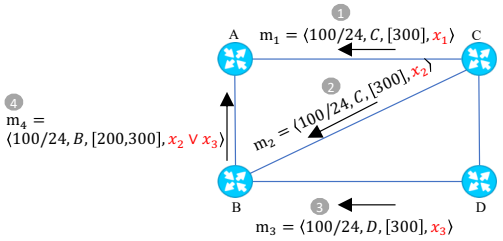


Figure 6: Demonstration of symbolic route simulation for the motivating example. Here a route advertisement message is represented as ⟨prefix, next hop, AS path, guard⟩, where the guard is highlighted in red.

C-E carries 100% of f_1's traffic, the STF of f_1 on the link is thus represented as $1 * x_1\overline{x_2}x_3$ (note that $\overline{x_2}$ is equivalent to $1 - x_2$).

Summing up all the symbolic formulas, we can succinctly represent the STF of f_1 on C-E in all five scenarios as $1 * x_1 + 0.5 * \overline{x_1}x_2x_3$. Note that there are three remaining failure scenarios not shown in Figure 1(a)-(e), and this formula does not cover them either.

4.3 Symbolic Traffic Execution Framework

Given a network with generated guarded RIBs and SR policies for all routers, YU then runs symbolic traffic execution for each flow entering the network. Below we describe the high-level framework of the algorithm and §4.4 discusses how to encode the forwarding process of a router.

At a high level, the symbolic traffic execution algorithm on a flow f runs in iterations, simulating the propagation of f's traffic. In each iteration, the algorithm updates the STF of f on each link by executing the forwarding process of each router given the incoming traffic computed in the last iteration. Thus, for a given link l, the i-th iteration computes the traffic fraction of f that can reach l in i hops in arbitrary failure scenarios. The algorithm terminates when a fixed point is achieved (*i.e.*, no update of STF on any link; not shown in Algorithm 1) or it reaches the maximum iteration number (which can be determined by the TTL [1] setting in practice).

To implement the above algorithm when no SR is configured, we can maintain a vector storing the STF of each link and then update the vector iteratively. However, when SR is configured, packets in the flow may be attached to different label stacks based on the SR path and forwarded differently. For example, router D in Figure 1(a) needs to attach two label stacks, namely [E, F] and [C, F], to packets in f_2; router E has to forward those labeled packets based on the labels. Thus, the algorithm must consider the various label stacks attached to the flow f, and maintain a matrix $\mathbb{M}$ mapping each link l and label stack $\mathcal{S}$ to a symbolic formula, representing the fraction of traffic of f on l when f has the label stack $\mathcal{S}$. Note that the number of label stacks is bounded by the sum of all SR paths' length, which is typically small in practice.

Algorithm 1: symbolic traffic execution

```
1  Function simulate(f):
2      M_0[l, S] ← 0 for all link l and label stack S;
3      construct a pseudo incoming link l^R to f's receiving router R;
4      M_0[l^R, ∅] ← 1;
5      for i = 1 to maximum iteration number I do
6          M_i[l, S] ← 0 for all outgoing link l and label stack S;
7          forall router R and label stack S do
8              ω_R^f ← sum{M_{i-1}[l, S]} for all incoming link l of R;
9              M_i ← M_i + forward(R, f, S, ω_R^f);
10     return M_I
```

Algorithm 1 shows the high-level symbolic traffic execution algorithm for one flow. For illustration, the algorithm maintains multiple copies of $\mathbb{M}$, denoted $\mathbb{M}_i$ for the i-th iteration. The algorithm first constructs a pseudo incoming link l^R to the router R where f enters the network, and initializes the matrix $\mathbb{M}_0$ by only assigning $\mathbb{M}_0[l^R, \emptyset]$ to 1, representing 100% of f's traffic at l^R with no labels attached (*i.e.*, $\emptyset$) on any packets (line 2-4). Then, the algorithm iteratively updates $\mathbb{M}$. In the i-th iteration, the algorithm updates $\mathbb{M}_i$ for all routers and label stacks (line 7-9) by computing the STF of f that router R receives (line 8) and executing the forwarding process of R (shown as the *forward* function, elaborated below) given the flow f, label stack $\mathcal{S}$, and received STF ω_R^f (line 9).

4.4 Symbolic Traffic Forwarding

We now discuss how to symbolically execute the forwarding process of a router in order to implement the *forward* function referenced above. First, we discuss how to encode several key forwarding primitives, including route selection, (weighted-)ECMP, and route iteration. Those encodings can be pre-computed and cached (with prefix classification) for high efficiency. Based on those encodings, we then describe the symbolic forwarding algorithm.

Encoding route selection. Given a guard RIB `gRIB`, traffic sending to *dstip* is forwarded to the next hops of selected routes; we use a boolean formula s_r^{dstip} to represent whether r is selected for the *dstip*. Without failures, selected routes are the most preferred routes in the RIB that match *dstip*. However, with failures, less preferred routes can also be selected. For example, in the guard RIB of router A (Figure 3), r_2 can be selected if r_1 does not present. Therefore, for a rule r, if *dstip* cannot match r, s_r^{dstip} is trivially 0; for a matching rule r, we need to ensure that all more preferred rules do not present and r should present. So,

$$s_r^{dstip} = \begin{cases} 0, \text{ if } dstip \text{ does not match } r \\ g_r \wedge \bigwedge_{r':dstip \text{ matches } r' \text{ and } r' \prec r} \overline{g_{r'}}, \text{ otherwise.} \end{cases}$$

Encoding (weighted-)ECMP. If there are multiple selected rules in a guarded RIB `gRIB` for traffic sending to *dstip*, the traffic is equally balanced among those rules. Given a rule r in `gRIB`, we encode the

ratio of such traffic that would be forwarded by r as a symbolic formula c_r^{dstip}, shown as

$$c_r^{dstip} = s_r^{dstip} / \sum_{r'} s_{r'}^{dstip},$$

where $\sum_{r'} s_{r'}^{dstip}$ encodes the total number of selected rules for *dstip*. Note that when $s_r^{dstip} = 1$ (*i.e.*, r is selected for *dstip*), the encoding of s_r^{dstip} ensures that all more preferred rules do not present and all less preferred rules are not selected; thus, $\sum_{r'} s_{r'}^{dstip}$ naturally encodes the number of equally preferred rules.

For a SR policy containing multiple weighted SR paths, traffic matching the policy is load-balanced proportional to the configured weights. We encode the ratio of the matching traffic forwarded on each path p as c_p, which is shown as

$$c_p = \frac{g_p * w_p}{\sum_{p'} g_{p'} * w_{p'}},$$

where $\sum_{p'} g_{p'} * w_{p'}$ encodes the total weights of valid paths based on the guard $g_{p'}$.

Encoding route iteration. When a flow f matches a route with an indirect next hop IP *nip*, it needs to resolve the direct next hop by looking up the IGP RIB or SR policies. For the IGP RIB, multiple directly connected links are resolved based on the selected rules. We use a vector $\mathbb{V}_{nip}^{\texttt{IGP}}$ to encode the ratio of traffic forwarded on each link resolved for *nip*, as

$$\mathbb{V}_{nip}^{\texttt{IGP}}[l] = \sum_{r: nh_r = l} c_r^{nip} \text{ for all outgoing link } l.$$

Thus, if the traffic that needs to forward to *nip* is V, the traffic that link l will forward is $V * \mathbb{V}_{nip}^{\texttt{IGP}}[l]$.

When a SR policy is matched for the flow f and the indirect next hop *nip*, multiple SR paths may be used for forwarding. For each SR path p, we need to further resolve the direct next hops for the first node in the path by looking up the IGP RIB. We use the vector $\mathbb{V}_p^{\texttt{SR}}$ to encode the ratio of traffic forwarded on each link resolved for p. Suppose *ip* is the address of the first node in p, then

$$\mathbb{V}_p^{\texttt{SR}}[l] = c_p * \mathbb{V}_{ip}^{\texttt{IGP}}[l] \text{ for all outgoing link } l.$$

Symbolic forwarding algorithm. With the encoding of key forwarding primitives, Algorithm 2 shows the symbolic forwarding of a flow f's traffic. At the top level, Function *forward* takes the router R, flow f, its potential label stack $\mathcal{S}$, and the f's symbolic traffic fraction ω as input, and computes a matrix $\mathbb{M}$, which stores the symbolic traffic fraction for each outgoing link and label stack. When the label stack is empty, the algorithm calls Function *forwardIp* to forward f based on its *dstip*; otherwise, it calls Function *forwardSr* to forward the f based on its label stack (line 2).

Due to space limit, we only explain Function *forwardIp* in detail. First, the function initializes the matrix $\mathbb{M}$ to be returned (line 4). Then for each rule r in the guarded RIB which can match f's *dstip*, it checks if the next hop of r is a direct next hop. If so, f would be forwarded to the associated link l without any label stacks (line 9); thus only $\mathbb{M}[l, \emptyset]$ is updated by adding the incoming fraction ω times the ECMP encoding c_r. Otherwise, it needs to resolve the indirect next hop and update $\mathbb{M}$ (line 7), using Function *resolveNhIp* which is based on the encoding of route iteration.

Algorithm 2: symbolic traffic forwarding on a router

```
1  Function forward(R, f, S, ω):
2      return S = ∅? forwardIp(R, f, ω) : forwardSr(R, f, S, ω);
3  Function forwardIp(R, f, ω):
4      M[l, S] ← 0 for all link l and label stack S;
5      forall rule r in gRIB_R s.t. f can match r do
6          if nh_r is an indirect next hop then
7              M ← M + resolveNhIp(R, f, nh_r, ω * c_r);
8          else
9              M[l, ∅] ← M[l, ∅] + ω * c_r where l = nh_r;
10     return M;
11 Function resolveNhIp(R, f, nip, ω):
12     if (f, nip) matches some SR policy P then
13         return M where M[l, S] = ω * Σ_{p∈P:p has label stack S} V_p^SR[l]
14     return M where M[l, ∅] = ω * V_nip^IGP[l] and M[l, S] = 0 for all link l
           and label stack S ≠ ∅;
15 Function forwardSr(R, f, S, ω):
16     let S = [R_1, R_2, ..., R_j];
17     if R = R_1 then
18         return forward(R, f, [R_2, ..., R_j], ω);
19     return M where M[l, S] = ω * V_ip^IGP[l] and M[l', S'] = 0 elsewhere,
           ip is the address of R_1;
```

Example. Consider the forwarding of f_2 on router D in Figure 1. For simplicity, we only consider the state of link D-E and use y as the symbolic variable for it. Figure 7 shows the (intermediate) results of the execution of Algorithm 2.

When f_2 reaches router D, Function *forwardIp* looks up the guarded RIB and finds the only (iBGP) rule with the indirect next hop to router F. Next, it uses Function *resolveNhIp* to resolve the indirect next hop, which further looks into the configured SR policy. For the SR path p_1, it needs to resolve the address of E using IGP, as E is the first node in the path. As shown in Figure 7(a), there are two IS-IS routes to E; the route selection encoding s and ECMP encoding c indicate that the traffic to E is forwarded via one of l_1 and l_2 entirely, which leads to the IGP route iteration encoding $\mathbb{V}_{10.0.0.5}^{\texttt{IGP}}$ for E, shown as the left vector in Figure 7(b). The middle and right vectors in Figure 7(b) show the route iteration encoding for the SR path p_1 and p_2, respectively (note that $c_{p_1} = 0.75$ and $c_{p_2} = 0.25$). Finally, *resolveNhIp* returns the matrix in Figure 7(c) which indicates that the STF of f_2 on l_1 (D-E) is $0.75y$ and that of l_2 (D-C) is $0.75\overline{y} + 0.25$, as expected.

4.5 Verifying TLPs

Given the computed matrix $\mathbb{M}_f$ by Algorithm 1 for each incoming flow f, the STL τ_l of link l can be computed as $\sum_{f,\mathcal{S}} V_f * \mathbb{M}_f[l, \mathcal{S}]$, where V_f is the traffic volume of f. Thus, to verify that l's traffic load is in a range $[v_1, v_2]$, we just need to check if there is a satisfiable solution to the *negation* of this property under the k-failure constraint, shown as

$$\left(\sum_i \overline{x_i} \le k \right) \wedge \left(\tau_l(x) > v_2 \vee \tau_l(x) < v_1 \right).$$

When a solution exists, there is a violating failure scenario where the TLP does not hold; otherwise, the TLP is verified under arbitrary k failures. We will describe the algorithm used to check if the above inequality is satisfiable in the next section (Theorem 5.1).

	Prefix	Next Hop	IGP Cost	Guard	s	c
r_1:	10.0.0.5/32	l_1	10005	y	y	y
r_2:	10.0.0.5/32	l_2	20005	1	$\overline{y}$	$\overline{y}$

(a) The (partial) guarded IS-IS RIB for router D annotated with route selection and ECMP encodings (s and c) for 10.0.0.5.

$$\begin{array}{c} \\ l_1 \\ l_2 \end{array} \begin{pmatrix} y \\ \overline{y} \end{pmatrix} \quad \begin{pmatrix} 0.75y \\ 0.75\overline{y} \end{pmatrix} \quad \begin{pmatrix} 0 \\ 0.25 \end{pmatrix}$$

(b) The route iteration encodings: (Left) $\mathbb{V}^{\mathsf{IGP}}_{10.0.0.5}$, (Middle) $\mathbb{V}^{\mathsf{SR}}_{p_1}$, (Right) $\mathbb{V}^{\mathsf{SR}}_{p_2}$.

$$\begin{array}{c} \\ l_1 \\ l_2 \end{array} \begin{array}{c} \mathcal{S}_1 \\ \begin{pmatrix} 0.75y \\ 0.75\overline{y} \end{pmatrix} \end{array} \begin{array}{c} \mathcal{S}_2 \\ \begin{pmatrix} 0 \\ 0.25 \end{pmatrix} \end{array}$$

(c) Output of Function *forward*, *forwardIp*, and *resolveNhIp*.

Figure 7: Illustrative execution of Algorithm 2. y is the symbolic variable for link D-E. l_1 and l_2 denote link D-E and D-C; $\mathcal{S}_1$ and $\mathcal{S}_2$ denote the label stack [E,F] and [C,F], respectively.

5 SYMBOLIC ENCODING SIZE REDUCTION

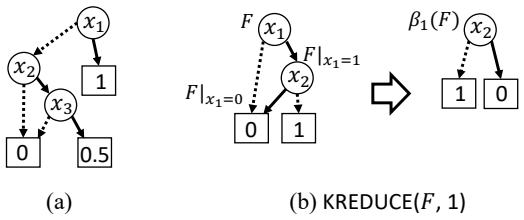
The symbolic traffic execution framework offers a general approach toward verifying TLPs. However, a naive implementation of this framework may induce significantly high overhead when applied to production-scale networks, due to the large number of failure scenarios and flows. In this section, we propose several optimization techniques that significantly improve the efficiency. First, we discuss our choice of MTBDD as the symbolic representation to concisely encode a large number of failure scenarios (up to $O(n^k)$). Second, we propose k-failure equivalence reduction optimization to reduce the size of MTBDD encodings. Last, we propose link-local flow-equivalence optimization that reduces the number of flows needed to be considered when verifying TLPs on one link.

5.1 Symbolic Representation as MTBDD

Like any symbolic execution framework, one of the keys to building a highly efficient implementation of the symbolic traffic execution framework is to compactly represent the symbolic encodings (*e.g.*, symbolic traffic load/fraction), while supporting efficient operations required in symbolic traffic execution (*e.g.*, multiplication and inverse of the symbolic encodings). Naive representations, such as SMT formulas, do not suffice in both the compactness and efficiency, thus are not suitable for our purpose.

As seen in §4, the symbolic encodings are pseudo-boolean functions. A compact representation of pseudo-boolean functions is well studied in formal verification, known as *multi-terminal binary decision diagrams* or MTBDDs [6, 15, 22].

Similar to a binary decision diagram (BDD) [12], a MTBDD is a single-source directed acyclic graph, but with multiple terminal nodes. As an example, consider the symbolic traffic fraction $1 * x_1 + 0.5 * \overline{x_1} x_2 x_3$ as shown in §4.2. Its MTBDD representation is shown in Figure 8(a), where each intermediate node is labeled with a variable while the terminal node with concrete numbers. A path from the source node to a terminal node corresponds to an assignment to all encountered variables on the path, where a dashed (solid, resp.) edge denotes assigning the variable to 0 (1, resp.), and the terminal node denotes the resulting number under that assignment. For example, when $x_1 = 0$ and $x_2 = x_3 = 1$, the corresponding path in the MTBDD reaches the terminal node 0.5, which represents that $1 * x_1 + 0.5 * \overline{x_1} x_2 x_3$ is evaluated to 0.5, corresponding to the failure scenario (d) in §4.2. For a MTBDD $\mathcal{F}$, we use $\mathcal{F}|_p$ to denote the sub-graph following the path p. For example, let $\mathcal{F}$ denote the MTBDD in Figure 8(a). Then $\mathcal{F}|_{x_1=0}$ denotes the sub-graph starting from variable x_2.



(a) (b) KREDUCE(F, 1)

Figure 8: MTBDD Examples

All the operations required in the symbolic traffic execution can be easily performed on MTBDDs, such as inverse (*e.g.*, $1/\tau$), arithmetic operations (*e.g.*, addition and multiplication), and equivalence checking of pseudo-boolean functions. For example, the inverse of a pseudo-boolean function can be simply performed by inverting the numbers on all terminal nodes of its MTBDD. Moreover, as we show next, verifying TLPs using MTBDD is significantly faster than using naive techniques (*e.g.*, SMT solving). Therefore, we use MTBDD as the symbolic representation in our implementation of the symbolic traffic execution framework.

5.2 k-Failure Equivalence Reduction

Even with the compact symbolic representation using MTBDD, the symbolic traffic execution may still incur high performance overhead. Specifically, during the iterative computation, the MTBDDs may grow as large as $O(2^n)$ (n is the network size), as the general symbolic traffic execution framework essentially encodes *all* possible failure scenarios, which is of the size $O(2^n)$.

However, for k-failure TLP verification, we only need to consider the scenarios with no more than k failures, which is significantly less (*i.e.*, $O(n^k)$ instead of $O(2^n)$), especially for practical cases where k is usually small. Thus, we need to reduce and simplify the MTBDDs along the symbolic traffic execution process by considering the k-failure specificity (*i.e.*, no more than k failures), while still yielding correct TLP verification results. As an example, suppose the STL of a link computed by symbolic traffic execution is $60 * x_1 + 25 * (\overline{x_1} x_2 + \overline{x_1} \overline{x_2} \overline{x_3})$. We can see that verifying any 2-failure TLPs for this link is equivalent to that of $60 * x_1 + 25 * \overline{x_1} x_2$, since failure of 3 links (*i.e.*, $\overline{x_1} \overline{x_2} \overline{x_3}$) is out of the verification requirement.

While it may seem obvious for the above example to reduce the symbolic representation of the STL by simply dropping $\overline{x_1} \overline{x_2} \overline{x_3}$ from the formula, it is not easy to reduce a MTBDD in the general case. For example, a naive approach may expand a MTBDD to a decision tree, prune all paths beyond the k-failure requirement (*i.e.*, representing more than k failures), and finally reduce it back. Such

an approach may successfully reduce the size while maintaining the correctness of verification, however, it induces exponentially high overhead, losing the benefit of employing MTBDD.

In the following, we discuss our approach to reduce MTBDD for k-failure TLP verification.

Definition 5.1 (k-failure equivalent MTBDDs). *Given two MTBDDs $\mathcal{F}$ and $\mathcal{G}$, each of them denotes a pseudo-boolean function ($\{0,1\}^n \rightarrow \mathbb{R}$) that maps a link failure scenario to a real number. We define them to be k-failure equivalent iff they always map to identical results when the number of simultaneous failures do not exceed k. We denote $\mathcal{F} \approx_k \mathcal{G}$ iff*

$$\forall x \in \{0,1\}^n, \sum_i \overline{x_i} \leq k \implies \mathcal{F}(x) = \mathcal{G}(x).$$

k-failure equivalent MTBDD REDUCE. Based on the above insight, we customize the implementation of MTBDD operations to yield k-failure equivalent results. Here, we focus on the REDUCE operation. Given an arbitrary MTBDD, REDUCE identifies and merges identical sub-structures in the graph to yield a compact representation for it [12, 22], lowering the overhead of using MTBDDs.

We follow this standard approach and extend it with the above insights (Definition 5.1) to define a k-failure equivalent KREDUCE $(\mathcal{F}, k)$ operation. We use $\beta_k(\mathcal{F})$ as a shorthand.

Definition 5.2 (Principles of KREDUCE). *Assume MTBDD $\mathcal{F}$'s source node represents variable x_i. We define $\beta_k(\mathcal{F})$ as*

$$\beta_0(\mathcal{F}) \triangleq \mathcal{F}(x_1 = 1, x_2 = 1, ..., x_n = 1), \quad (1)$$

$$\beta_k(c) \triangleq c, \text{ where } c \text{ is a terminal node}, \quad (2)$$

and that

$$\beta_k(\mathcal{F}) \triangleq \begin{cases} \beta_k(\mathcal{F}|_{x_i=1}), \text{ if } \beta_{k-1}(\mathcal{F}|_{x_i=1}) = \beta_{k-1}(\mathcal{F}|_{x_i=0}) & (3) \\ x_i * \beta_k(\mathcal{F}|_{x_i=1}) + \overline{x_i} * \beta_{k-1}(\mathcal{F}|_{x_i=0}), \text{ otherwise.} & (4) \end{cases}$$

There are two main extensions on top of the standard REDUCE operation. First, as highlighted in (1), a MTBDD is reduced to a terminal node if there is no more failure allowed ($k = 0$). Second, as highlighted in (3), two $(k-1)$-failure equivalent sub-graphs can be merged even when they are not isomorphic (more detailed discussions are in Appendix A). Based on the principle, we implemented the KREDUCE operation using dynamic programming techniques, with the complexity proportional to the size of $\mathcal{F}$ and k.

Example. Figure 8(b) illustrates the KREDUCE operation with $k = 1$. The original MTBDD, $\mathcal{F}$, represents the function $1 * x_1\overline{x_2}$. Since $\beta_0(\mathcal{F}|_{x_1=0}) = 0$ and $\beta_0(\mathcal{F}|_{x_1=1}) = \mathcal{F}|_{x_1=1,x_2=1} = 0$, KREDUCE $(\mathcal{F}, 1)$ merges two equivalent sub-graphs and yields the result $1 * \overline{x_2}$, which is 1-failure equivalent to $\mathcal{F}$.

Correctness of k-failure equivalent verification. Employing k-failure equivalent MTBDD operations yields correct verification results due to the following properties. For any MTBDD $\mathcal{F}$,

Lemma 1 (KREDUCE yields a k-equivalent MTBDD).

$$\text{KREDUCE}(\mathcal{F}, k) \approx_k \mathcal{F}.$$

Lemma 2 (Any path in KREDUCE $(\mathcal{F}, k)$ contains no more than k failures). *Assume an arbitrary path p in KREDUCE $(\mathcal{F}, k)$ encodes variable assignments for $x_{p_1}, x_{p_2}, ..., x_{p_m}$, then*

$$\sum_i \overline{x_{p_i}} \leq k.$$

Detailed proofs are in Appendix A.

Theorem 5.1 (Adopting k-failure reduced operations yield correct TLP verification results). *Using k-failure reduced operations, we conduct symbolic traffic execution (§4.2) to obtain the symbolic fraction matrix $\mathbb{M}_f^*$ for each flow f, then compute the symbolic traffic load $\tau_l^* = \sum_{f,\mathcal{S}} V_f * \mathbb{M}_f^*[l, \mathcal{S}]$ for a link l. To verify that l's traffic load is in a range $[v_1, v_2]$, it suffices to check the existence of a counter-example $y \in \{0,1\}^n$ such that*

$$\tau_l^*(y) > v_2 \vee \tau_l^*(y) < v_1. \quad (5)$$

The intuition is that, symbolic traffic execution preserves the k-failure equivalence between $\mathbb{M}_f^*$ and $\mathbb{M}_f$ during each iteration, and further, between τ_l^* and τ_l. Since τ_l^* encodes all valid failure scenarios in τ_l (Lemma 1) and no more other mappings (Lemma 2), the above checking suffices. See Appendix B for a formal proof.

Efficient TLP verification. Given Theorem 5.1, checking a TLP can be performed by simply checking the values of all terminal nodes of the MTBDD of τ_l^*, which is significantly more efficient compared to naive algorithms such as SMT solving.

5.3 Link-local Flow-Equivalence Reduction

As shown above, verifying a TLP on a link requires to aggregate the symbolic traffic load (represented as MTBDDs) for all flows on the link. A naive approach may apply m MTBDD addition operations, where m is the total number of flows on that link, which is prohibitively inefficient because MTBDD addition across flows can cause MTBDD sizes to explode (see an example in Figure 18 in Appendix), and there can be billions of flows in production WANs.

To mitigate such overhead, we leverage the link-local equivalence of different flows. Despite the differences in global forwarding behavior, two flows may distribute the same fraction of traffic on a link in every failure scenario. For example, f_1 and f_2 (Figure 1) behave differently in general, but their distribution along the link E-F are always identical. This allows us to combine them as a single flow when computing τ_{E-F}, instead of spreading them arbitrarily into a series of heavy additions.

Moreover, the MTBDD representation allows highly efficient grouping of flows based on the link-local equivalence. For any two flows f_1 and f_2, they are link-local equivalent on a link l if their MTBDDs of corresponding symbolic traffic fractions are equivalent, *i.e.*, $\omega_l^{f_1} = \omega_l^{f_2}$.

Assume that there are p link-local equivalent classes of flows on a link l and let G_i be the set of all flows in the i-th class, f_i^* be any flow in G_i. The total symbolic traffic load can be efficiently computed with p MTBDD additions as

$$\tau_l = \sum_{i=1}^{p} \left(\omega_l^{f_i^*} * \sum_{f:f \in G_i} V_f \right).$$

6 DEPLOYMENT AND USE CASES

We built YU based on the approaches in §4 and §5 with additional optimization heuristics such as global flow equivalence [39] and pruning-based early termination for property violations.

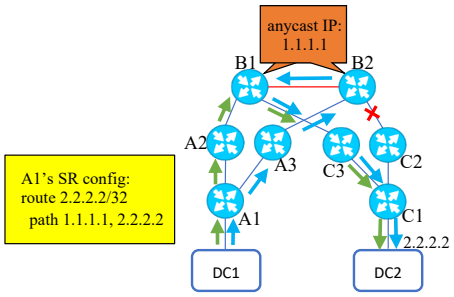


Figure 9: Link overload due to vulnerable SR configuration.

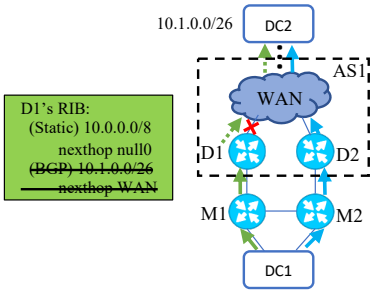


Figure 10: Service traffic dropping due to misconfiguration.

YU has been deployed in our production WAN (with > 1000 routers) for months and used regularly to check the failure tolerance of the WAN against multiple traffic load properties, including link overloading and significant traffic dropping/increasing. YU has successfully identified many failure scenarios that may lead to severe network reliability issues. Below, we share two representative, real cases, covering potential outages including service quality downgrade and service unavailability.

Overloading due to vulnerable SR features. Given the complexity of our WAN in both the scale and network protocols, it used to be hard for our operators to analyze if any link would be overloaded when some link failed due to unexpected maintenance or breakdown. Fortunately, with YU, our operators can now automatically analyze the overloading of *all* links under *arbitrary* link failures, and identify potential vulnerabilities before they trigger network outages. Figure 9 shows one such vulnerability case YU identified. In this case, a large amount of service traffic is forwarded from DC1 in one region to DC2 in another remote region. To fully utilize the bandwidth of paths between the two regions, our operator configured a SR policy that steers the traffic from DC1 to DC2 via two backbone routers (*i.e.*, B1 and B2) in the middle. To make the configuration easy to maintain, the operator configured an anycast IP address on B1 and B2 (*i.e.*, 1.1.1.1 in the example) and specifies one SR path based on that IP. The intention of this configuration was to establish two SR tunnels, namely A1-A2-B1-C3-C1 (annotated with green arrows in Figure 9) and A1-A3-B2-C2-C1, to forward the traffic. When one tunnel breaks due to link failures, another tunnel still has enough capacity to carry the entire traffic. However, YU found a violation. When link B2-C2 failed, a new tunnel A1-A3-B2-B1-C3-C1 (annotated with blue arrows in Figure 9) would be established in order to satisfy the required segment A1-B2 as specified in the SR configuration. As a result, a large amount of traffic would be forwarded in the new tunnel including a low-capacity link B1-B2, causing that link to be overloaded. We report this issue to the operation team and get confirmed that this issue is caused by the vulnerable SR configuration, which is hard to be detected without the help of YU.

Service traffic dropping due to misconfiguration. Our WAN is designed with enough redundancy to tolerate any single-point failures. However, due to misconfiguration on routers, service traffic may still be unexpectedly dropped when a failure happens, causing severe service outages. Figure 10 shows such a potential outage YU identified. In this case, DC1 sends a large volume of traffic to a service in DC2 with the prefix 10.1.0.0/26. Without any failures, the traffic can be forwarded along two paths, annotated with green and blue arrows in Figure 10. However, YU identified a significant traffic drop of DC2 when the router D1's link to the WAN failed, despite that there are many redundant paths left. The root cause of this potential outage is due to BGP misconfiguration on router D1. Particularly, D1 configured a default static route that intentionally drops traffic matching 10/8. D1 redistributed that route into BGP and advertised it to router M1 without advertising other more specific routes. In the normal scenario, traffic would match the 10/8 route on M1 and be forwarded to D1, which forwards the traffic to WAN using the 10.1/26 route. However, when the D1-WAN link failed, the 10.1/26 route would not present on D1, so traffic sent from M1 to D1 would match the 10/8 route and get dropped (D2 has similar configurations, so traffic would not be forwarded from M1 to M2). Again, without YU this issue is hard to detect given the large network size and high complexity of configurations. Fortunately, YU detected this issue and prevented a severe service outage.

7 EVALUATION

In this section, we evaluate YU's performance on three main questions: **(Q1)** How efficient is YU in handling large-scale networks and flows? **(Q2)** How does the performance of YU compare to state-of-the-art works? **(Q3)** How do the design choices and optimizations in YU contribute to its efficiency?

7.1 Production WAN Benchmarking

First, we evaluate the performance of YU in our production networks. We selected four networks for analysis: the entire WAN and three sub-networks named N0, N1, and N2. These sub-networks represent small, medium, and large scales, respectively. Key characteristics of these networks are summarized in Table 3. All four networks are configured with BGP, IS-IS, and SR. We conducted the experiments on a server with a 96-core 2.40GHz processor and 791 GB RAM.

Scalability to the network size (Q1). We evaluate YU's performance on the four networks for both router and link failures with $k \in [1, 4]$, excluding the combinations where the k-failure route simulation cannot terminate within 12 hours. We provide YU with flows entering the network within a one-hour time window and

[3]All numbers are approximate for confidentiality reasons.

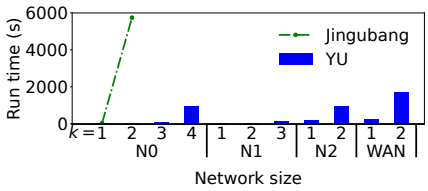


Figure 11: Verification time for k-link failures. Jingubang takes 95.7 min for N0 with $k = 2$, which is 448× slower than that of YU.

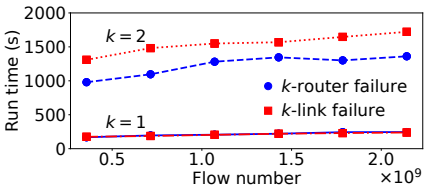


Figure 12: Verification time for the WAN. With a 6× increase in the flow number, verification time only increases by 31.5% for 2-link failures.

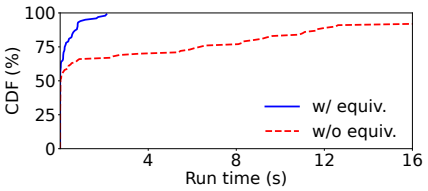


Figure 13: Link-local equivalence reduces the 90th percentile time of TLP verification on all links from 12.51 s to 0.79 s (16×) for 1-link failures.

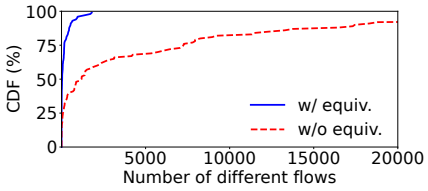


Figure 14: Equivalence lowers flow numbers on all links by 33× in 90th percentile for 1-link failures.

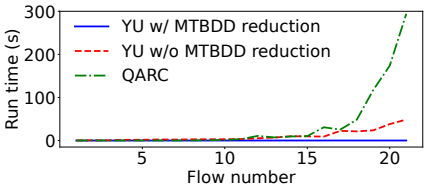


Figure 15: QARC costs 4.9 min for 2-link failures with 21 input flows, which is 2042× slower than YU.

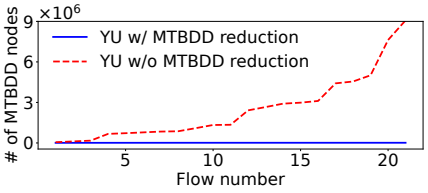


Figure 16: `KREDUCE` cuts MTBDD node numbers down by 627× for 2-link failures with 21 input flows.

Table 3: Network characteristics.[3]

	# Routers	# Links	# Prefixes	# Flows (1 hour)
N0	100	200	3×10^3	5×10^7
N1	200	500	2×10^6	2×10^8
N2	500	2500	2×10^6	2×10^9
WAN	1000	4000	2×10^6	2×10^9

employ it to verify if any links will become overloaded in any failure scenario. Figure 11 depicts the verification time for link failures (the results for router failures are similar, as shown in Figure 17 in Appendix). We observe for the small (N0) and medium (N1) networks (with hundreds of links), YU can complete verification within minutes for $k = 4$ and $k = 3$, respectively. Notably, even for the entire WAN with thousands of links, YU can finish verification within only 28.6 min for $k = 2$, which is far beyond the requirement for practical deployment.

Comparison with Jingubang (Q2). We compare YU's performance with that of Jingubang based on the same setting above. To use Jingubang, we must exhaustively examine all concrete failure scenarios so that we can utilize its incremental simulation to verify if any link becomes overloaded. However, this method takes > 1 hour (and > 1 day) for incremental traffic simulation (and incremental route simulation) in N1 with $k = 2$. As a result, larger networks are not feasible for comparing the performance of Jingubang with YU under different k values. Consequently, we restrict our analysis to the small network, N0. As shown in Figure 11, Jingubang requires 95.7 min for incremental traffic simulation in enumerated failure cases for $k = 2$, which is 448× slower than YU's verification time.

Scalability to the number of flows (Q1). We next assess the scalability of YU by varying the scale of input flows on the entire WAN. By adjusting the time window from 7 min to 60 min, the number of flows entering our WAN during this period increases from $\sim 4 \times 10^8$ to $\sim 2 \times 10^9$. The verification time of YU under these flow inputs and different failure scenarios are presented in Figure 12. Despite a 6× increase in the number of flows, the verification time for 2-link (2-router) failures only increase from 21.8 min (16.3 min) to 28.7 min (22.7 min), marking a mere 31.5% (39.0%) increment. We delve deeper into this phenomenon and attribute it to YU's capacity to identify flow equivalence, both globally and locally, significantly enhancing its scalability.

Effects of equivalence (Q3). We now show the effectiveness of link-local equivalence reduction on the entire WAN. Figure 13 displays the TLP verification time CDF of randomly selected 100 links with and without the equivalence reduction for 1-link failures. As depicted in the figure, the 90th percentile time is substantially reduced from 12.51 s to 0.79 s by applying link-local equivalence. For $k = 2$, YU could take hours to verify TLPs for a single link without link-local equivalence reduction, making it impractical to verify thousands of links in production. Additionally, we present the flow numbers on each link before and after the equivalence reduction in Figure 14, where the 90th percentile flow number decreases from $\sim 1.7 \times 10^4$ to ~ 500, aligning with the previous results.

7.2 Synthetic Dataset Benchmarking

We then evaluate YU in a series of networks called `FT-`m. `FT-`m employs the FatTree topology [5] with m pods, consisting of $O(m^2)$ routers operating under the eBGP protocol and $O(m^3)$ links. The links between the aggregation routers and the core (edge) routers have bandwidths of 100 Gbps (40 Gbps). We introduce pairwise flows between edge routers with volumes of 5 Gbps, using YU to verify the absence of overloaded links during arbitrary 2-link

Table 4: Verification time (in seconds). We inject 4%, 8%, 12%, and 16% of the pairwise flows between the edge routers to FT-4, FT-8, and FT-12, and use YU, QARC, and Jingubang to verify whether any links will be overloaded under arbitrary 2-link failures.

Network	FT-4				FT-8				FT-12			
Flow number	2 (4%)	5 (8%)	7 (12%)	9 (16%)	40 (4%)	79 (8%)	119 (12%)	159 (16%)	204 (4%)	409 (8%)	613 (12%)	818 (16%)
YU	**0.057**	**0.103**	**0.143**	**0.123**	**0.709**	**1.197**	**1.631**	**1.759**	**22.28**	**51.948**	**116.059**	**230.484**
QARC [52]	0.142	0.145	0.196	0.839	2.848	3.404	6.074	74.427	556.866	460.410	> 3600	3559.669
Jingubang [39]	0.392	0.478	0.464	0.472	38.201	41.327	43.894	45.466	1655.358	1854.675	2054.381	2252.567

failures. The experiments were conducted on a machine equipped with a 16-core 4.70GHz processor and 32 GB of memory.[4]

Comparison with QARC (Q2). QARC lacks support for, and cannot be readily extended to, iBGP and SR protocols, which are essential for our production WAN. Consequently, we are unable to evaluate QARC in our WAN. Instead, we conducted experiments in FT-4 to compare the performance between QARC and YU with respect to the number of flows. As shown in Figure 15, QARC's verification time increases exponentially with the number of flows. Even for just 21 input flows, QARC requires 4.9 min to complete verification, making it 2042× slower than YU. We also evaluated YU's performance under different network sizes. Table 4 shows that YU is 42.3× (15.4×) faster than QARC in FT-8 (FT-12) with 16% of all pairwise flows between edge routers. For reference, we also list Jingubang's verification time for the same network in the table.

Effects of KREDUCE (Q3). Without KREDUCE, YU is unable to complete verification for any of our production networks within an hour, even with just a single input flow. Therefore, we investigated how KREDUCE contributes to YU's efficiency in FT-4. As shown in Figure 15, the verification time for YU escalates rapidly when k-failure MTBDD reduction is disabled, even for a small set of input flows. For instance, with 21 input flows, YU without KREDUCE takes 49 s to complete verification, whereas YU with KREDUCE only requires 0.14 s. We then present the number of MTBDD nodes generated during verification in Figure 16. In the aforementioned example, KREDUCE lowers the number of MTBDD nodes from $> 4 \times 10^6$ to $< 2 \times 10^4$, aligning with the verification time.

8 RELATED WORK

Network verification in general. There is rich literature in checking the correctness of networks. On the device level, formal verification [16, 42, 53] and test generation techniques [44, 48, 63] have been developed to check the correctness of individual devices' forwarding behaviors. On the network scale, various verification techniques have been proposed to check the control plane [4, 8, 17, 20, 23, 24, 33, 46, 47, 50, 55–57, 59] and the data plane [9, 29, 31, 32, 34, 35, 43, 45, 51, 54, 58], mostly regarding reachability properties (*e.g.*, routes/packets reachability) across the network.

Quantitative property analysis. Jingubang [39] and QARC [52] are the closest to YU, which we have discussed in detail in §2.1. YU's focus on traffic properties is also different from the work on probabilistic analysis of the network control plane [26, 50, 61] and data plane [21, 49]. YU verifies traffic at the network implementation level, different from the work [13, 14, 62] validating networks at the design level.

Applications of BDD in network analysis. BDD has been recognized as a useful data structure in network analysis. Existing work utilize BDD to encode various networking components, such as packet fields [7, 11, 60] or route announcements [17, 25, 26], which differs from YU's use of MTBDD. Similar to YU, SRE [61] uses BDD to encode up to k failures. However, SRE cannot represent quantitative properties nor can it verify TLPs.

Traffic engineering (TE). Most TE work [18, 28, 30, 38] optimizes traffic allocation from a global view of the network, which is not applicable under our current WAN settings. However, for TE implemented by reconfiguring routers or injecting routes [10, 37, 40], YU can serve as a complementary tool by verifying that no TLPs will be violated when applying TE decisions.

Network emulation. Current emulator systems, *e.g.*, CrystalNet [41], mainly focus on mimicking the network behaviors. Similar to Jingubang, they are not practical to check k-failure traffic property within one-run. Another issue of emulators is the cost. Ye *et al.* [57] have discussed the expensive cost of emulator systems prevents them from being used for global WAN checking.

9 CONCLUSION

This paper presented YU, the first traffic load property verification system for arbitrary k-failure scenarios applied to a global production WAN. YU proposed a general symbolic traffic execution framework that can support a wide range of network features (*e.g.*, BGP, IS-IS, SR) and novel optimizations (*i.e.*, MTBDD size reduction and link-local flow equivalence) to improve the efficiency. YU has been deployed in production and successfully identified several failure-tolerance issues. YU's verification is efficient and can scale to production WANs.

ACKNOWLEDGMENTS

We thank our shepherd and the SIGCOMM reviewers for their insightful comments. This work is supported by National Key Research and Development Plan, China (Grant No. 2023YFB2903902) and Alibaba Cloud through Alibaba Research Intern Program and Alibaba Innovative Research Program. Ennan Zhai and Chenren Xu are the co-corresponding authors. Ruihan Li and Chenren Xu are affiliated with School of Computer Science at Peking University, Zhongguancun Laboratory, and Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

[4]We use a separate evaluation setup due to restrictions of YU's deployment environment.

REFERENCES

[1] 1981. Internet Protocol. RFC 791. (Sept. 1981). https://doi.org/10.17487/RFC0791

[2] 1989. Border Gateway Protocol (BGP). RFC 1105. (June 1989). https://doi.org/10.17487/RFC1105

[3] 1990. OSI IS-IS Intra-domain Routing Protocol. RFC 1142. (Feb. 1990). https://doi.org/10.17487/RFC1142

[4] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1402958.1402967

[6] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebric decision diagrams and their applications. *Formal methods in system design* 10 (1997), 171–206.

[7] Ryan Beckett and Aarti Gupta. 2022. Katra: Realtime Verification for Multilayer Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 617–634. https://www.usenix.org/conference/nsdi22/presentation/beckett

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[9] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 328–341. https://doi.org/10.1145/2934872.2934909

[10] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/3341302.3342069

[11] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3603269.3604866

[12] Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (Aug 1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[13] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. 2020. Lancet: Better network resilience by designing for pruned failure sets. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '20)*. Association for Computing Machinery, New York, NY, USA, 53–54. https://doi.org/10.1145/3393691.3394195

[14] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. 2017. Robust Validation of Network Designs under Uncertain Demands and Failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 347–362. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/chang

[15] E. M. Clarke, K. L. McMillan, X Zhao, M. Fujita, and J. Yang. 1993. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th International Design Automation Conference (DAC '93)*. Association for Computing Machinery, New York, NY, USA, 54–60. https://doi.org/10.1145/157485.164569

[16] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 950–965. https://doi.org/10.1145/3519939.3523715

[17] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz

[18] Mikel Jimenez Fernandez and Henry Kwok. 2017. Building Express Backbone: Facebook's new long-haul network. (2017). https://en.wikipedia.org/w/index.php?title=Traffic_engineering_(transportation)&oldid=1169174329

[19] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. 2018. Segment Routing Architecture. RFC 8402. (July 2018). https://doi.org/10.17487/RFC8402

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[21] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–309.

[22] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design* 10, 2 (01 Apr 1997), 149–169. https://doi.org/10.1023/A:1008647823331

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[24] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient Verification of Network Fault Tolerance via Counterexample-Guided Refinement. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 305–323.

[25] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: an intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 958–973. https://doi.org/10.1145/3385412.3386019

[26] Nick Giannarakis, Alexandra Silva, and David Walker. 2021. ProbNV: probabilistic verification of network control planes. *Proc. ACM Program. Lang.* 5, ICFP, Article 90 (aug 2021), 30 pages. https://doi.org/10.1145/3473595

[27] Peter L Hammer, I Rosenberg, Sergiu Rudeanu, et al. 1963. On the determination of the minima of pseudo-Boolean functions. *Studii si Cercetari matematice* 14 (1963), 359–364.

[28] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[29] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[30] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2486001.2486019

[31] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094

[32] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies*. Technical Report MSR-TR-2014-102. Microsoft. https://www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/

[33] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRooT: Proactive Verification of DNS Configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. https://doi.org/10.1145/3387514.3405871

[34] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[35] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 15–27. https://www.usenix.org/conference/

nsdi13/technical-sessions/presentation/khurshid
[36] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[37] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. 2022. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 325–338. https://www.usenix.org/conference/nsdi22/presentation/krishnaswamy
[38] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2785956.2787478
[39] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. 2024. Reasoning about Network Traffic Load Property at Production Scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1063–1082. https://www.usenix.org/conference/nsdi24/presentation/li-ruihan
[40] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 527–538. https://doi.org/10.1145/2619239.2626314
[41] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 599–613. https://doi.org/10.1145/3132747.3132759
[42] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical verification for programmable data planes. In *ACM SIGCOMM (SIGCOMM)*.
[43] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes
[44] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark W. Barrett, and Peter Athanas. 2018. p4pktgen: Automated Test Case Generation for P4 Programs. In *Symposium on SDN Research (SOSR)*.
[45] Aurojit Panda, Katerina Argyraki, Mooly Sagiv, Michael Schapira, and Scott Shenker. 2015. New Directions for Network Verification. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset (Eds.), Vol. 32. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 209–220. https://doi.org/10.4230/LIPIcs.SNAPL.2015.209
[46] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 699–718. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths
[47] B. Quoitin and S. Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE Network* 19, 6 (2005), 12–19. https://doi.org/10.1109/MNET.2005.1541716
[48] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 136–151. https://doi.org/10.1145/3603269.3604834
[49] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor meets Scott: semantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 557–571. https://doi.org/10.1145/3009837.3009843
[50] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 750–764. https://doi.org/10.1145/3387514.3405900
[51] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 314–327. https://doi.org/10.1145/2934872.2934881
[52] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2020. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 974–988. https://doi.org/10.1145/3385412.3385976
[53] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. 2021. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/3452296.3472937
[54] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. https://doi.org/10.1145/3341302.3342088
[55] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some Complexity Results for Stateful Network Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 811–830.
[56] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2012. FSR: Formal Analysis and Implementation Toolkit for Safe Interdomain Routing. *IEEE/ACM Transactions on Networking* 20, 6 (2012), 1814–1827. https://doi.org/10.1109/TNET.2012.2187924
[57] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217
[58] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. 2020. NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 181–200. https://www.usenix.org/conference/nsdi20/presentation/yuan
[59] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 575–589. https://www.usenix.org/conference/nsdi20/presentation/zhai
[60] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 241–255. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng
[61] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 336–349. https://doi.org/10.1145/3544216.3544264
[62] Yunmo Zhang, Hong Xu, Chun Jason Xue, and Tei-Wei Kuo. 2022. Probabilistic Analysis of Network Availability. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940438
[63] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: scalable network testing for programmable data planes. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 350–364. https://doi.org/10.1145/3544216.3544247

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A PROPERTIES OF KREDUCE OPERATION

Lemma 1: KREDUCE yields a k-equivalent MTBDD

$$\mathsf{KREDUCE}(\mathcal{F}, k) \approx_k \mathcal{F}.$$

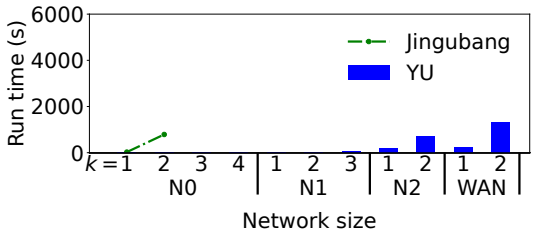


Figure 17: Verification time for k-router failures. YU requires only 22.3 min on average for the entire WAN with $k = 2$.

Proof. We follow Observation 5.2 to reason about KREDUCE's behavior.

The above property holds for the base case, when $k = 0$ or when $\mathcal{F}$ represents a terminal node c.

For the recursive case, assume that the above property is satisfied by smaller inputs, *i.e.*,

$$\beta_k(\mathcal{F}|_{x_i=1}) \approx_k \mathcal{F}|_{x_i=1},$$

$$\beta_{k-1}(\mathcal{F}|_{x_i=0}) \approx_{k-1} \mathcal{F}|_{x_i=0},$$

$$\beta_{k-1}(\mathcal{F}|_{x_i=1}) \approx_{k-1} \mathcal{F}|_{x_i=1}.$$

For each x such that $\sum_i \overline{x_i} \le k$, the evaluation of $\mathcal{F}(x)$ breaks down depending on the assignment of x_i. We also denote $y = x_1 ... x_{i-1} x_{i+1} ... x_n$. Thus,

$$\begin{aligned}\mathcal{F}(x) &= x_i * \mathcal{F}|_{x_i=1}(x) + \overline{x_i} * \mathcal{F}|_{x_i=0}(x)\\ &= x_i * \mathcal{F}|_{x_i=1}(y) + \overline{x_i} * \mathcal{F}|_{x_i=0}(y).\end{aligned}$$

We consider the following two cases.

(1) When $\beta_{k-1}(\mathcal{F}|_{x_i=1}) = \beta_{k-1}(\mathcal{F}|_{x_i=0})$, KREDUCE $(\mathcal{F}, k)$ returns $\beta_k(\mathcal{F}|_{x_i=1})$.
If $x_i = 1$,

$$\begin{aligned}\beta_k(\mathcal{F})(x) &= \beta_k(\mathcal{F}|_{x_i=1})(x)\\ &= \mathcal{F}|_{x_i=1}(x) && \text{by } k\text{-equivalence}\\ &= \mathcal{F}(x).\end{aligned}$$

If $x_i = 0$,

$$\begin{aligned}\beta_k(\mathcal{F})(x) &= \beta_k(\mathcal{F}|_{x_i=1})(x)\\ &= \beta_k(\mathcal{F}|_{x_i=1})(y) && x_i \text{ is irrelevant}\\ &= \mathcal{F}|_{x_i=1}(y) && k\text{-equivalence}\\ &= \beta_{k-1}(\mathcal{F}|_{x_i=1})(y) && (k-1)\text{-equivalence}\\ &= \beta_{k-1}(\mathcal{F}|_{x_i=0})(y)\\ &= \mathcal{F}|_{x_i=0}(y) && (k-1)\text{-equivalence}\\ &= \mathcal{F}(x).\end{aligned}$$

(2) Otherwise, KREDUCE $(\mathcal{F}, k)$ reduces its two successor subgraphs, and the property holds.

□

Lemma 2: KREDUCE bounds the number of failures in each path Let $\mathcal{P}(\text{KREDUCE}(\mathcal{F}, k))$ denote the set of all paths from the source node to a terminal node in the MTBDD KREDUCE$(\mathcal{F}, k)$. Assume an arbitrary path $p \in \mathcal{P}(\text{KREDUCE}(\mathcal{F}, k))$ encodes variable assignments for $x_{p_1}, x_{p_2}, ..., x_{p_m}$, and we use $\|\bar{p}\|$ to denote the number of zeros within these variable assignments $(\sum_i \overline{x_{p_i}})$, then

$$\|\bar{p}\| \le k.$$

Proof. When $k = 0$, the result of KREDUCE$(\mathcal{F}, k)$ is a terminal node and does not contain any variable assignment. The above property holds.

For the recursive case, assume that the property holds for smaller inputs, *i.e.*,

$$\forall p \in \mathcal{P}(\beta_k(\mathcal{F}|_{x_i=1})), \quad \|\bar{p}\| \le k, \tag{6}$$

$$\forall p \in \mathcal{P}(\beta_{k-1}(\mathcal{F}|_{x_i=0})), \quad \|\bar{p}\| \le k-1. \tag{7}$$

We consider the following cases.

(1) When $\beta_{k-1}(\mathcal{F}|_{x_i=1}) = \beta_{k-1}(\mathcal{F}|_{x_i=0})$, KREDUCE $(\mathcal{F}, k)$ returns $\beta_k(\mathcal{F}|_{x_i=1})$.
According to induction Hypothesis (6), any path p in $\beta_k(\mathcal{F}|_{x_i=1})$ satisfies $\|\bar{p}\| \le k$.

(2) Otherwise, KREDUCE $(\mathcal{F}, k)$ reduces its two successor subgraphs and returns $x_i * \beta_k(\mathcal{F}|_{x_i=1}) + \overline{x_i} * \beta_{k-1}(\mathcal{F}|_{x_i=0})$. For any path $p \in \mathcal{P}(\text{KREDUCE}(\mathcal{F}, k))$, we denote p as $x_i :: p_{tail}$, since the source node corresponds to x_i.
If $x_i = 1$,

$$\begin{aligned}\|\bar{p}\| &= 0 + \|\overline{p_{tail}}\| && p_{tail} \in \mathcal{P}(\beta_k(\mathcal{F}|_{x_i=1}))\\ &\le \text{k.} && \text{Hypothesis (6)}\end{aligned}$$

If $x_i = 0$,

$$\begin{aligned}\|\bar{p}\| &= 1 + \|\overline{p_{tail}}\| && p_{tail} \in \mathcal{P}(\beta_{k-1}(\mathcal{F}|_{x_i=0}))\\ &\le 1 + \text{k - 1} && \text{Hypothesis (7)}\\ &\le \text{k.}\end{aligned}$$

Thus, the property holds. □

B CORRECTNESS OF k-FAILURE REDUCED VERIFICATION

This section proves that employing k-failure equivalent MTBDD operations (*e.g.*, KREDUCE) throughout the verification yields correct results.

Lemma 3 (Symbolic traffic execution yields k-failure equivalent symbolic fractions). *Given an arbitrary flow f, symbolic traffic execution (§4.3) computes the symbolic fraction $\mathbb{M}_f[l, \mathcal{S}]$ for each link l. By employing k-failure equivalent MTBDD operations, it computes $\mathbb{M}^*_f[l, \mathcal{S}]$ instead.*

Then,

$$\mathbb{M}_f[l, \mathcal{S}] \approx_k \mathbb{M}^*_f[l, \mathcal{S}].$$

Proof. The symbolic traffic execution is carried out by iterations. Starting from a same $\mathbb{M}_0[l, \mathcal{S}]$, every step is composed of k-failure equivalent MTBDD operations, which preserve the k-failure equivalence of inputs and outputs.

Assume that after the completion of the $(i-1)$-th iteration, the induction hypothesis holds, *i.e.*,

$$\forall l, \mathcal{S}, \mathbb{M}_{i-1}[l, \mathcal{S}] \approx_k \mathbb{M}^*_{i-1}[l, \mathcal{S}].$$

Now we consider the computation of $\mathbb{M}^i_i$. For example, on line 8 in Algorithm 1, we have

$$\omega^f_R \approx_k \omega^{*f}_R.$$

Similarly, all other computations yield k-failure equivalent results such that

$$\forall l, \mathcal{S}, \mathbb{M}_i[l, \mathcal{S}] \approx_k \mathbb{M}^*_i[l, \mathcal{S}].$$

By induction, this lemma holds. □

Lemma 4 (k-failure equivalent symbolic fraction leads to k-failure equivalent symbolic traffic loads).

Proof. Given k-failure equivalent symbolic fractions $\mathbb{M}^*_f$, the total symbolic traffic load on a link l is computed as

$$\tau^*_l = \sum_{f,\mathcal{S}} V_f * \mathbb{M}^*_f[l, \mathcal{S}].$$

Since k-failure equivalent MTBDD operations (*e.g.*, multiply, summation) preserve the results' k-failure equivalence, we have

$$\tau^*_l \approx_k \tau_l.$$

□

Theorem 5.1: adopting k-failure reduced operations yields correct TLP verification results Using k-failure reduced operations, we conduct symbolic traffic execution (§4.2) to obtain the symbolic fraction matrix $\mathbb{M}^*_f$ for each flow f, then compute the symbolic traffic load $\tau^*_l = \sum_{f,\mathcal{S}} V_f * \mathbb{M}^*_f[l, \mathcal{S}]$ for a link l. To verify that l's traffic load is in a range $[v_1, v_2]$, it suffices to check the existence of a counter-example such that

$$\left(y \in \{0,1\}^n\right) \wedge \left(\tau^*_l(y) > v_2 \vee \tau^*_l(y) < v_1\right).$$

Proof. By the above lemmas, we have

$$\forall l, \tau_l \approx_k \tau^*_l,$$

where τ_l is computed following §4.5.

(1) If there is indeed a failure scenario $x^a \in \{0,1\}^n$ that leads to TLP violation on link l, *i.e.*,

$$\left(\sum_i \overline{x^a_i} \le k\right) \wedge \left(\tau_l(x^a) > v_2 \vee \tau_l(x^a) < v_1\right).$$

Let $y^a = x^a$. By Definition 5.1, $\tau_l(y^a) = \tau^*_l(y^a)$ must hold. Thus,

$$\left(y^a \in \{0,1\}^n\right) \wedge \left(\tau^*_l(y^a) > v_2 \vee \tau^*_l(y^a) < v_1\right).$$

Such a y^a is easily detectable by traversing the terminal nodes of τ^*_l. Thus, the theorem holds.

(2) If the TLP always holds, *i.e.*,

$$\forall x \in \{0,1\}^n, \left(\sum_i \overline{x^a_i} \le k \implies v_1 \le \tau_l(x^a) \le v_2\right).$$

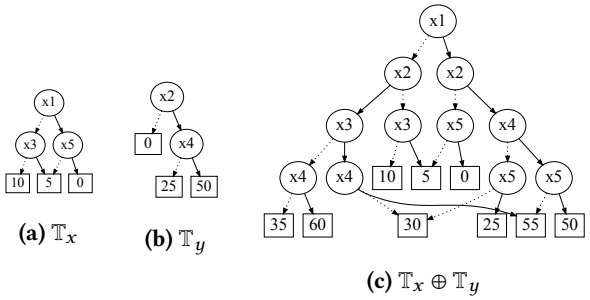
For any path p in τ^*_l, it encodes an assignment y. By Lemma 2, $\sum_i \overline{y_i} \le k$ must hold. This entails $\tau^*_l(y) = \tau_l(y)$. Thus,

$$v_1 \le \tau^*_l(y) \le v_2.$$

Since any terminal node in τ^*_l is within the range $[v_1, v_2]$, the theorem holds. □

C LINK-LOCAL FLOW EQUIVALENCE REDUCTION

Figure 18 shows that adding two small MTBDDs leads to a large MTBDD, resulting in MTBDD size explosion.



(a) $\mathbb{T}_x$ **(b)** $\mathbb{T}_y$ **(c)** $\mathbb{T}_x \oplus \mathbb{T}_y$

Figure 18: Summation of two MTBDDs can lead to significantly larger size. In a MTBDD, dashed lines denote link failures and solid lines denote normal links.