Symbolic Router Execution

Peng Zhang
Xi'an Jiaotong University

Dan Wang
Xi'an Jiaotong University

Aaron Gember-Jacobson
Colgate University

Abstract

Network verification often requires analyzing properties across different spaces (header space, failure space, or their product) under different failure models (deterministic and/or probabilistic). Existing verifiers efficiently cover the header or failure space, but not both, and efficiently reason about deterministic or probabilistic failures, but not both. Consequently, no single verifier can support all analyses that require different space coverage and failure models. This paper introduces Symbolic Router Execution (SRE), a general and scalable verification engine that supports various analyses. SRE symbolically executes the network model to discover what we call packet failure equivalence classes (PFECs), each of which characterises a unique forwarding behavior across the product space of headers and failures. SRE enables various optimizations during the symbolic execution, while remaining agnostic of the failure model, so it scales to the product space in a general way. By using BDDs to encode symbolic headers and failures, various analyses reduce to graph algorithms (e.g., shortest-path) on the BDDs. Our evaluation using real and synthetic topologies show SRE achieves better or comparable performance when checking reachability, mining specifications, etc. compared to state-of-the-art methods.

CCS Concepts

• **Computer systems organization** → *Reliability*.

Keywords

network verification, equivalence classes, symbolic execution

ACM Reference Format:
Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic Router Execution. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544264

1 Introduction

Network verifiers enable operators to proactively reason about a network's forwarding behaviors to avoid potential problems. For example, verifiers can compute all-pairs reachability for a specific failure scenario to verify a planned router outage will not compromise reachability [1]; analyze a specific prefix under probabilistic link failures to verify a proposed configuration change directs the traffic through a waypoint 99.9% of the time [21]; or examine all prefixes under a bounded number of link failures to verify a proposed configuration refactoring will not jeopardize security [22].

Such verification tasks require network verifiers to:
(1) Reason about a network's behavior across a large header space and/or failure space. For example, the header space can be as large as 2^{104} when considering 5-tuples, and the failure space can be as large as 2^l when considering all possible combinations of link failures in a network with l links.
(2) Reason about a network's behavior under different failure models. For example, verifying a property holds under a bounded number of failures (k) requires a deterministic failure model where each node or link can either be up or down but the total number of failures is bounded by k; whereas verifying a property holds with high probability (e.g., 99.9%) requires a probabilistic failure model where nodes or links can fail according to some distribution.

To meet the first requirement, existing verifiers exploit similarity across packet headers and failure scenarios. Some verifiers [6, 10, 11, 20, 27] divide the header space into *packet equivalence classes (PECs)*, such that headers belonging to the same PEC traverse the same forwarding path under a specific failure scenario. Other verifiers [3, 5, 12–15, 21, 24, 26] divide the failure space into *failure equivalence classes (FECs)*, such that failure scenarios belonging to the same FEC result in the same forwarding path for a specific source-destination pair. Analyzing one packet from each PEC or one failure from each FEC is sufficient to characterize the network's behavior across the entire header or failure space, respectively.

However, due to the dual influence of headers and failures on forwarding path selection, PECs may not be the same across failure scenarios and FECs may not be the same across headers. Consequently, verifiers leveraging PECs must independently analyze every failure scenario of interest, and verifiers leveraging FECs must independently analyze every prefix of interest. This causes the verifiers to scale poorly to the product space of headers and failures (§8).

To meet the second requirement, different verifiers target different failure models and use different optimizations to efficiently explore the failure space. For example, checking a property under a deterministic failure model only requires exploring the boundary of failure scenarios—e.g., the minimum number of failures (k) that violate the property; verifiers do not need to consider scenarios with more than k failures, even if the property is satisfied in these scenarios, and many verifiers are optimized to identify the boundary without evaluating every scenario with fewer than k failures [3, 5, 12–14, 24, 26]. On the other hand, checking a property under a probabilistic failure model requires exploring every failure scenario, and summing the probabilities of scenarios that satisfy the property; verifiers are optimized to explore the most likely failure scenarios first, and stop exploring when an acceptable confidence threshold is reached [15, 21]. Consequently, verifiers designed for deterministic

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544264

failure models are not amenable to probabilistic failures, and verifiers designed for probabilistic failure models are inefficient when applied to deterministic failures.

In summary, existing verifiers lack *scalability* to reason about the product space of headers and failures, and *generality* to efficiently handle both deterministic and probabilistic failure models. The lack of scalability is due to verifiers not jointly reasoning about headers and failures to explore their dual impact on forwarding behaviors. The lack of generality stems from verifiers binding to the failure model too early (i.e., when the network model is created).

This paper introduces *Symbolic Router Execution (SRE)*, a *general* and *scalable* verification engine that can explore the product space of headers and failures while remaining agnostic of the failure model. In a nutshell, SRE applies a variant of symbolic execution tailored to configuration verification, by viewing the network as a program (model) consisting of a control plane and a data plane, and making headers and failures symbolic when executing the model.

Why SRE is scalable. First, SRE symbolically executes the network control *and* data planes to account for the correlation among headers and failures and uncover Packet Failure Equivalence Classes (PFECs)—header and failure combinations for which a specific forwarding path is used. SRE executes a control plane model with symbolic failures (i.e., link states) to derive the FEC for each route, and SRE executes a data plane model where forwarding rules include FECs as another matching field (in addition to IP prefix) so FECs "carry over" to the data plane. In this way, failures and headers jointly determine a set of forwarding paths, each of which corresponds to an equivalence class in the product space (i.e., PFECs). Second, SRE significantly reduces computations by: (*i*) using Binary Decision Diagrams (BDDs) [4] to represent symbolic headers and failures, and (*ii*) applying three optimizations—route pruning, prefix pruning, and abstract interpretation.

Why SRE is general. First, PFECs capture all possible forwarding paths of all packets under all possible failures, such that any property related to packets and their forwarding paths—e.g., reachability, waypointing, isolation, or load balancing—can be analyzed. Second, SRE is agnostic of the failure model when symbolically executing the network model. The failure model is only specified when analyzing properties. Delaying the binding to failure model allows SRE to efficiently support deterministic and probabilistic failure models in a general way. Interestingly, using BDDs also makes SRE more general: we can analyze properties through graph algorithms. For example, analyzing a property's failure tolerance reduces to computing the shortest path on a BDD, and analyzing the probability of a property holding reduces to computing a weighted sum on a BDD.

In summary, this paper makes the following contributions:

- We introduce symbolic router execution (SRE), a configuration verification engine that scales to the product space of headers and failures, and generalizes to different failure models.
- We design and implement SRE and apply various optimizations to make it scalable and fast. We implement three types of analyses on top of SRE to demonstrate its generality in supporting different types of analyses.

Failure model	Equivalence classes: PECs	Equivalence classes: FECs
Determ.	Batfish [11], Plankton [20], ERA [10], ShapeShifter [6], DNA [27], Config2Spec [8]	Config2Spec [8], ARC [12], Tiramisu [3], Hoyan [26], Minesweeper [5], Origami [13], Bagpipe [24], NV [14]
Prob.		NetDice [21], ProbNV [15]

Table 1: Equivalence classes and failure models supported by existing network configuration verifiers

- We use real and synthetic topologies to show SRE achieves better or comparable performance to state-of-the-art methods when checking properties, mining specifications, computing probabilities, etc.

2 Motivation

In this section, we discuss common network management tasks (§2.1), and the scalability and generality limitations of existing network verifiers with respect to these tasks (§2.2).

2.1 Tasks

Some common network management tasks require reasoning about a network's behavior across a large product space of headers and failures under both deterministic and probabilistic failure models. For example:

Verifying changes. Verifying a configuration change has the desired effect (e.g., restricting access to a prefix) only requires analyzing the targeted header space(s). However, checking for unintended side-effects is harder, because changes may impact seemingly unrelated header spaces: e.g., augmenting a route filter with a high priority rule that blocks routes with certain community tags may overshadow a rule that permits routes for certain prefixes. Consequently, verifying a change is side-effect free requires checking all (manually-specified or mined) requirements, which often span *a large portion of the header and failure spaces* [7, 8] and include *both deterministic and probabilistic failure tolerances* [21].

Mining network requirements. Many configuration verifiers assume operators can clearly specify what to verify—e.g., a router should (not) be able to reach a certain prefix. However, network requirements are rarely explicitly documented. Consequently, researchers have developed network specification miners [7, 8, 17], which check several types of forwarding properties (e.g., reachability, isolation, and waypoint traversals) for the *entire header space under a large range of failure scenarios* (e.g. all single- and dual-link failures), to mine specific requirements implied by router configurations. It is also desirable to generalize these requirements to groups of prefixes [17] and soft failure tolerance levels (e.g., "four 9s" availability) [21], which requires reasoning about *both deterministic and probabilistic failures*.

Some common management tasks may not require reasoning about the product space of headers and failures or multiple failure models, but it is desirable to construct a "one-size-fits-all" verifier that accommodates these tasks as well.

2.2 Related Work

Existing verifiers lack the scalability and generality required to conduct that aforementioned tasks which reason about a network's behavior across a large product space of headers and failures under

both deterministic and probabilistic failure models. As summarized in Table 1, existing verifiers compute either packet equivalence classes (PECs) or failure equivalence classes (FECs)—which do not extend to the product space—and accommodate either deterministic or probabilistic failure models—which require different explorations of the failure space and different optimizations.

PECs or FECs. PECs and FECs allow verifiers to exploit similarity in network forwarding behaviors across packet headers or failure scenarios, respectively. Batfish [11] and ERA [10] implicitly compute PECs, whereas Plankton [20] and DNA [27] explicitly compute PECs. Conversely, FECs are implicitly computed by: NetDice [21], which identifies "cold" links whose failure does not impact forwarding paths for a specific source-destination pair; Hoyan [26] and ProbNV [15], which identify link conditions that influence the existence/selection of a specific route; ARC [12] and Tiramisu [3], which compute path characteristics that are invariant across failures for specific source-destination pairs; and Minesweeper [5] and Bagpipe [24], which rely on an SMT solver's ability to learn equivalences in a domain-agnostic manner. Analyzing one packet from each PEC or one failure from each FEC is sufficient to characterize the network's behavior across the entire header or failure space, respectively.

However, since PECs and FECs may differ across failure scenarios and headers, respectively, verifiers leveraging PECs must independently analyze every failure scenario, and verifiers leveraging FECs must independently analyze every prefix. Thus, existing verifiers scale poorly to the product space of headers and failures (§8). The trade-off is illustrated by Config2Spec [8], which dynamically switches between a verifier that uses PECs [11] and a verifier that uses FECs [5] to reduce the work required to cover the product space.

Deterministic or probabilistic. Verifiers designed to reason about a bounded number of failures (k) are not directly amenable to probabilistic failures, and vice versa. For example, ARC [12] and Tiramisu [3] model a network's control plane as a graph and compute the min-cut to determine the minimum number of simultaneous link failures (k) under which a property (e.g., reachability) does not hold. However, they cannot compute the probability of properties because they do not consider link failures that exceed k but also preserve the property. On the other hand, NetDice [21] explores all failure scenarios by iteratively failing links and checking whether the property holds (until reaching a certain level of confidence in the probability a property holds), and applies a customized optimization to reduce the search space. However, exploring all failure scenarios is expensive and unnecessary when considering a bounded number of deterministic failures.

3 Overview

SRE is a general and scalable network verification engine which supports various analyses that require reasoning about a network's forwarding behavior across a large space of headers and failures and various failure models. In the following, we present the basic idea of SRE and show the workflow of SRE with an example.

3.1 Basic Idea

SRE is inspired by symbolic execution and its application in network verification.

Symbolic execution of programs. Symbolic execution [18] is a way to *abstractly* execute a program by making the inputs symbolic. When the symbolic executor encounters a conditional branch (e.g., if-else statement), it executes each branch and updates the *path condition*, which is a set of constraints encoding the branching decisions during the execution. As a result, symbolic execution explores each execution path at most once, and can discover equivalence classes of inputs (encoded by the path conditions). Generally, symbolic execution suffers from *path explosion*, and leverages many optimizations to mitigate it.

Symbolic execution of network control plane or data plane. Several network verifiers apply symbolic execution. HSA [16] can be viewed as symbolic execution over the data plane: it forwards packets with symbolic headers to discover PECs. Hoyan [26] can be viewed as symbolic execution over the control plane: it simulates the control plane with symbolic link states to discover FECs. However, as discussed earlier (§2.2), PECs and FECs do not extend to the product space of headers and failures.

Symbolic execution of network control plane and data plane. SRE symbolically executes the network control *and* data planes to exploit the correlation in forwarding behaviors among headers *and* failures. First, SRE executes a control plane model, where failures (i.e., link states) are symbolic, to derive the FEC for each route. Then SRE executes a data plane model, where both headers and failures are symbolic. When symbolically executing the data plane, SRE makes FECs another matching field (in addition to IP prefix) in forwarding rules, so that the FECs discovered during control plane execution "carry over" to the data plane. In other words, during data plane execution, failure scenarios and packet headers jointly determine a set of forwarding paths, each of which corresponds to an equivalence class in the product space, which we call packet failure equivalence classes (PFECs).

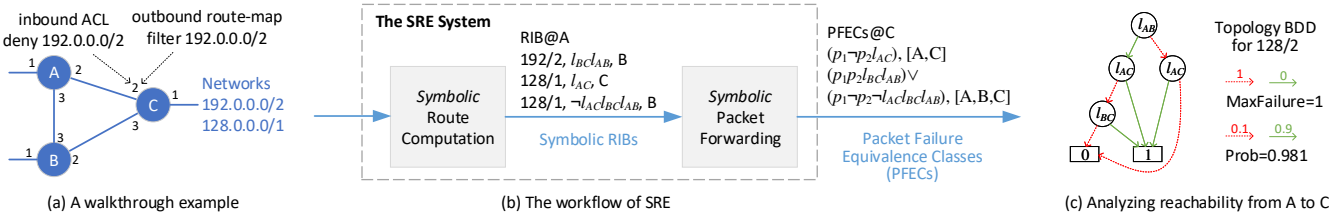
3.2 Workflow of SRE

An example network. Figure 1(a) shows an example network with three routers (A, B, and C) running BGP. Router C is connected to the network 128.0.0.0/1, and announces this prefix, as well as a longer prefix 192.0.0.0/2. The operator has a policy that all traffic to 192.0.0.0/2 should go through router B. Consequently, the operator configures port 2 on router C with: (1) an outbound route-map that prevents 192.0.0.0/2 from being advertised to A, and (2) an inbound ACL that blocks packets for 192.0.0.0/2 arriving from A.

At a high level, SRE consists of two steps: (1) symbolic route computation, and (2) symbolic packet forwarding. We use the above example to walk through these two steps.

(1) Symbolic route computation (SRC) takes configurations and topology as input, and computes symbolic RIBs, one for each router. Unlike a concrete RIB which maintains the current best routes, a symbolic RIB maintains *all* routes that may become the best route when links and/or nodes fail. SRC represents the state of each link with a boolean variable (1=up, 0=down),[1] and associates a *topology*

[1] Node failures are modeled as a combination of link failures.



(a) A walkthrough example (b) The workflow of SRE (c) Analyzing reachability from A to C

Figure 1: The walkthrough example, the workflow of SRE, and property analysis with SRE.

condition—a boolean formula consisting of these variables—with each route in the symbolic RIB to encode the failure scenarios under which the route becomes the best route.

Figure 1(b) shows the symbolic RIB at router A. For prefix 128/1, there are two best routes (second and third entries). The route with next hop C and topology condition l_{AC} becomes the best route if l_{AC} is up, while the route with next hop B and topology condition $\neg l_{AC} l_{BC} l_{AB}$ becomes the best route if l_{AC} is down but both l_{AB} and l_{BC} are up.

To generate symbolic RIBs, SRE initializes the topology condition of originated routes to symbolic value `True` (representing any combination of link failures). Then, SRE simulates the control plane, and during the simulation constrains the topology condition of each route. Unlike Hoyan [26] which encodes the topology condition using SAT constraints, SRE uses Binary Decision Diagrams (BDDs) [4], thus avoiding topology condition explosion for large networks (§8).

Binary Decision Diagram (BDD). As shown in Figure 1(c), a BDD is a rooted, directed acyclic graph (DAG) with two *terminal* nodes 0 and 1, and several (non-terminal) *decision* nodes. Each decision node corresponds to a boolean variable (l_{AB}, l_{BC}, etc.), and has two outgoing edges: a *dashed* edge and a *solid* edge, representing the boolean variable being assigned `False` and `True`, respectively. For example, in this BDD, the root node represents variable l_{AB}; if l_{AB} = `False`, we follow the dashed edge to another node representing variable l_{AC}, and if l_{AC} = `True`, we follow the solid edge to terminal node 1. A path from the root to the terminal 1 represents a truth assignment, e.g., l_{AB} = `False`, l_{AC} = `True` in this example.

(2) Symbolic packet forwarding (SPF) takes the symbolic RIBs as input, forwards symbolic packets through the network, and generates a set of PFECs. Each PFEC consists of all packet-failure tuples for which a forwarding path is used. Figure 1(b) shows the two PFECs whose forwarding paths are from A to C . The first PFEC $(p_1\neg p_2 l_{AC}, [A, C])$ represents packets 128/2 and failure scenarios where l_{AC} is up (other links can be up or down) for which the path $A{\rightarrow}C$ is used. Similarly, the second PFEC represents packets and failures for which the path $A{\rightarrow}B{\rightarrow}C$ is used.

To generate PFECs, SRE converts the symbolic RIBs into *symbolic FIBs*, where each FIB entry matches both the prefix and the topology condition of the corresponding route. Then, SPF augments packet headers with a topology condition, initializes the augmented header with a symbolic value of `True` (encoding all possible packet-failure tuples), and injects it at each router in the network. When a symbolic packet matches a FIB entry, SRE constrains the topology condition and destination IP of the packet with the topology condition and prefix, respectively, of the FIB entry.

Through SRC and SPF, SRE jointly explores the header space and failure space, in a way that is agnostic of the specific verification tasks (e.g., checking failure tolerance). This allows SRE to efficiently support a variety of analyses.

3.3 Property Analysis with SRE

SRE enables three types of analyses over various properties (e.g., reachability, waypointing, isolation, load balancing):

(1) *failure tolerance:* compute the maximum number of failures that a property can tolerate;
(2) *probabilistic:* estimate the probability that a property holds under probabilistic failures;
(3) *differential:* check for differences in failure tolerance/probability of a property after a configuration change.

While existing verifiers are targeted at one type of analysis, SRE enables all of these analyses based on the abstraction of PFECs. The reason is that SRE is agnostic of the analyses and outputs PFECs which collectively represent all possible forwarding behaviors (i.e., end-to-end forwarding paths), as well as the packets and failures for each behavior. Moreover, since each PFEC is encoded with a BDD (a graph), SRE allows operators to perform the analyses directly on top of BDDs with graph algorithms, agnostic of complex network semantics (e.g., routing protocols).

We use reachability as an example to show how computing failure tolerance and probabilities reduce to standard graph problems on top of BDDs. §6 discusses more analyses.

Example 1: Computing failure tolerance. Suppose operators need to know the failure tolerance for reachability of packets 128/2 from A to C. There are two PFECs at C satisfying the property, one traversing $A{\rightarrow}C$, and the other traversing $A{\rightarrow}B{\rightarrow}C$. We can compute a disjunction of these two PFECs, and extract the sub-BDD encoding the failures—which we call a *topology BDD*—as shown in Figure 1(c). In this topology BDD, the minimum number of dashed edges to the terminal node 0 is two, which corresponds to the minimum number of failures that violate the reachability property. That is, the maximum number of failures the reachability of packets 128/2 from A to C can tolerate is one less than the minimum number $(2 - 1 = 1)$. *Therefore, the problem of computing failure tolerance reduces to the problem of finding the shortest path: assign weight 1 to dashed edges and weight 0 to solid edges; compute the shortest path length k from the root node to terminal node 0; the failure tolerance is $k - 1$.*

Example 2: Computing probabilities. Suppose operators instead need to know the probability for reachability of packets 128/2 from A to C. Different from computing failure tolerance, which only cares about a failure scenario where the property does not hold with the minimum number of link failures, computing probability requires finding all failure scenarios where the property holds, and summing up their probabilities. In the topology BDD, each truth assignment (a path from the root to the terminal node 1) represents a set of failure scenarios where the property holds. *Therefore, the problem of computing probability reduces to the problem of finding all paths to a node on a graph, and computing a weighted sum of these paths.* For illustration purpose, assume each link fails independently with probability p (see §6 for details on dependent link failures or node failures), and assign weight p to dashed edges, weight $(1-p)$ to solid edges, weights 0 and 1 to the terminal nodes 0 and 1, respectively. Then, the probability is the weight of the root node, which can be computed recursively from the terminal node 1, according to: the weight of each node is the weighted sum of its two child nodes. In this example, we can easily see the probability is $0.9 * (0.9 + 0.1 * 0.9) + 0.1 * 0.9 = 0.981$.

4 Symbolic Route Computation

This step symbolically simulates the control plane to generate a *symbolic RIB* for each router. Each symbolic RIB consists of all possible routes that can materialize (i.e., become the best route) when links and/or nodes fail, and the corresponding failure scenarios.

4.1 Defining Symbolic Route

Before introducing symbolic routes, we first define *link variables*. For each link x in the network, its link variable is defined as boolean variable l_x, such that l_x = `1(True)/0(False)` means the link x is up/down, respectively. A *symbolic route* is a 2-tuple $(route, tc)$, where: $route$ is a concrete route for a specific protocol (e.g., OSPF and BGP) specifying the prefix, the next hop, and other protocol-specific attributes (e.g., AS Path); and tc, which stands for *topology condition*, is a predicate over link variables specifying the failure scenarios when $route$ becomes the best route.

Taking Figure 2(b) as an example, A will receive a route for prefix 128.0.0.0/1 from C, if link AC is up. Thus, SRE updates the topology condition of this route at router A to l_{AC}, where l_{AC} is a boolean variable denoting the state of link AC, i.e., $l_{AC} = 1$ or 0 if link AC is up or down, respectively.

SRE uses Binary Decision Diagrams (BDDs [9]) to encode topology conditions. Compared to SAT constraints, BDDs concisely encode boolean formulas and efficiently support conjunctions, disjunctions, and negations. Moreover, using BDDs allows SRE to support various analyses using graph algorithms (§6).

4.2 Computing Symbolic Routes

SRE computes symbolic routes by executing a control plane model, where each router repeatedly executes three steps: (1) import routes from neighboring routers; (2) rank all imported routes with existing routes and install the best routes into its RIB; and (3) export the best routes to neighboring routers. The execution terminates when the RIBs of routers do not change (i.e., a fixed point is reached).

Importing Routes. Initially, each router imports all routes declared in the configurations. Each such route has a tc = `True`. During recursive route computation, each router imports the routes exported (advertised) by neighboring routers, and filters or modifies routes according to routing policies. During this process, the topology condition is unchanged.

Ranking Routes. When a router receives multiple routes for the same prefix, it ranks these routes according to their priorities, and updates their topology conditions. Suppose there are n routes $r_1, r_2, \cdots, r_n$ with decreasing priority. The topology condition of r_k, denoted as $r_k.tc$, is updated by negating the topology conditions of all higher-priority routes, i.e., $r_k.tc = (\bigwedge_{i<k}(\neg r_i.tc)) \wedge r_k.tc$. For example, in Figure 2(c), router A receives another route r for 128.0.0.0/1 from router B, which has a topology condition of $r.tc = l_{BC}l_{AB}$. Assuming router A prefers routes with the shortest path, A will rank r lower than the one directly received from C, whose topology condition is l_{AC}. A updates $r.tc = \neg l_{AC}l_{BC}l_{AB}$.

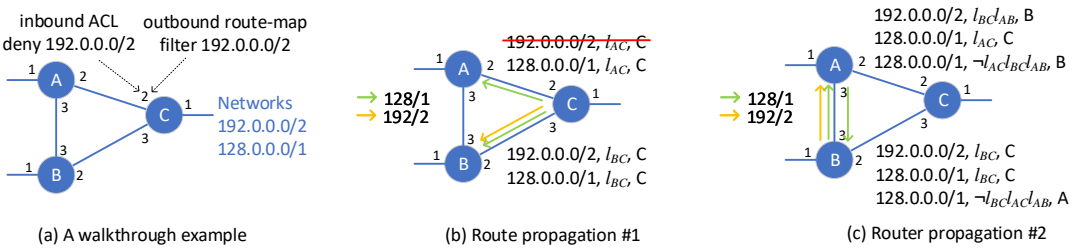
Exporting Routes. Each router exports to its neighbors the routes whose topology conditions are not `False`. The routes will first be filtered/modified according to the routing polices. For example, in Figure 2(b), router C filters the route 192.0.0.0/2 to be exported to router A according to the route map. For each route r exported by router R to its neighboring router N, $r.tc$ is updated to $r.tc \wedge l_{RN}$, where l_{RN} is the link between R and N.

Supporting multiple protocols. When there are multiple protocols (BGP, OSPF, static), SRE ranks routes first according to the administrative distance of their protocols, and then considers the protocol-specific priorities. When there are route dependencies—e.g., iBGP relies on OSPF to establish neighbor relationships—SRE will first compute the topology conditions for data plane reachability among iBGP peers (see §5), and then use the conditions as the link conditions among iBGP peers. That is, SRE views the connections among iBGP as virtual links whose conditions are computed based on data plane reachability analysis of OSPF.

Supporting route aggregation. For BGP, a router can use route aggregation to aggregate multiple routes of specific prefixes into a single route of summarizing prefix. When at least one route with a specific prefix is received, the aggregated route will be generated and advertised instead of the received route. This can introduce correlations among routes of different prefixes. It is easy to see that the link condition for the aggregated route is the disjunction of link conditions of all received routes whose prefixes are more specific.

Supporting multi-path routing. When multi-path routing (e.g., ECMP) is enabled, multiple routes for the same prefix may have the same priority, and these routes should all be selected as the best route. SRE realizes this by storing all routes for the same prefix in a two-dimensional list: each entry of the two-dimensional list is a list of routes with the same priority. When updating the topology condition of a route, SRE only negates the topology conditions of routes belonging to lists whose priorities are higher than the route.

Handling new higher-priority routes. A critical issue is dealing with new higher-priority routes. Specifically, when R imports a route whose priority is higher than some existing routes in its RIB, the topology conditions of these lower-priority routes become obsolete, and R should withdraw and re-advertise these routes.



(a) A walkthrough example (b) Route propagation #1 (c) Router propagation #2

Figure 2: The process of symbolic route computation.

This can trigger cascaded updates at all routers importing those lower-priority routes [26]. To be more efficient when handling new higher-priority routes, SRE only re-advertises lower-priority routes whose topology conditions change, without withdrawing any routes. To achieve this, SRE uses two topology conditions for each route: tc^{in} which represents the topology condition when the route is imported, and tc^{rib} which represents the topology condition when the route is inserted in to the RIB. When a set of routes are imported by a router, SRE will re-compute tc^{rib} for each route r_k whose priorities are equal or lower than any newly inserted routes, according to:

$$r_k.tc^{rib} = (\bigwedge_{1 \le i \le k-1} (\neg r_i.tc^{in})) \wedge r_k.tc^{in} \quad (1)$$

If $r_k.tc^{rib}$ is changed, we advertise it to all neighbors of R. Each advertised route r'_k will have $r'_k.tc^{rib}$ = `False`, and $r'_k.tc^{in} = r_k.tc^{rib} \wedge l$, where l is the variable of the link connected to the neighbor. The algorithm for computing the symbolic RIBs can be found in Appendix A.

5 Symbolic Packet Forwarding

5.1 Defining Symbolic Packets

SRE augments packet headers with a *topology condition*, which captures the failure scenarios under which the packet is forwarded. Suppose the original packet header has n bits (e.g., $n = 104$ for 5 tuple), and the network has m links; SRE uses a bit vector of length $(n + m)$ for the packet header. SRE symbolically executes the data plane by making the packet header symbolic and forwarding it through the network.

5.2 Generating symbolic FIBs

For each router, SRE generates a symbolic FIB, which is an ordered list of forwarding rules. Each forwarding rule is a 2-tuple $(match, port)$, where $match$ is a predicate (boolean formula) over packet headers and failure scenarios. For example, for symbolic route $(192/2, l_{BC}l_{AB}, B)$ at router A (Figure 3(a)), we will generate a forwarding rule $(p_1p_2l_{BC}l_{AB}, port3)$, where $p_1, p_2, \ldots, p_{32}$ are boolean variables for IP addresses (from the highest bit to the lowest bit), l_{AB} is a boolean variable for link AB, and port 3 is the port (interface) connected to router B (Figure 3(b)). Without loss of generality, we assume forwarding rules are ordered by prefix length (longest prefix has highest priority). For rules with the same prefix length, the priority is determined by the priority of their corresponding routes.

5.3 Computing predicates

After generating symbolic FIBs, we can forward symbolic packets through the network by matching forwarding rules in the FIBs. Each rule can be seen as a branch statement (e.g., if-then-else) in computer programs. However, each router often has a large number of rules, making the matching very inefficient. Therefore, we adopt the approach of pre-computing *port predicates* [25]. A port predicate is a boolean formula encoding the set of packets forwarded to a specific port (forwarding predicates), or allowed by a specific port (ACL predicates). Since there are a relatively small number of ports at each router (compared to the number of rules), matching based on port predicates will be more efficient.

Forwarding predicates. The forwarding predicate of a port is computed as a disjunction of the "effective" match fields of all rules which forward to that port. Here, "effective" means the match fields that are not overridden by higher-priority rules. For example, the effective match fields for the second rule p_1l_{AC} in the symbolic FIB of router A are $p_1l_{AC}\neg(p_1p_2l_{BC}l_{AB}) = p_1p_2l_{AC}\neg(l_{BC}l_{AB}) \vee p_1\neg p_2l_{AC}$, as shown in Figure 3(b). Here, the first term is for 192/2, which will match both the first and second rule. According to the priorities, the second rule will be matched only when the first rule is not matched, i.e., when either l_{BC} or l_{AB} is down. Since only the second rule forwards to port 2, then the forwarding predicate of port 2 is the effective match fields of the second rule. For another example, the port predicate for port 3 can be computed as a disjunction of the "effective" match fields of the first and third rules.

ACL predicates. Each port may have ACLs filtering inbound or outbound traffic. Therefore, we compute inbound and outbound ACL predicates for each port. The computation is similar to forwarding predicates. Returning to our example in Figure 3(b), router C has an ACL at port 2 filtering inbound packets for prefix 192/2. The inbound ACL predicate of port 2 is computed as $\neg(p_1p_2)$.

5.4 Forwarding packets

After computing predicates, we construct a symbolic packet matching all packet headers and failure scenarios (a logical `True` over header and link variables), and inject it at each router of the network. For each port of the router, we replicate the symbolic packet, and let it traverse the port. Suppose a port has a forwarding predicate P_1^{fwd}, an outbound ACL predicate P_1^{out}, and is connected to another port with an inbound ACL predicate P_2^{in}, through a link l, then we constrain the symbolic packet pkt by computing: $pkt \leftarrow pkt \wedge P_1^{fwd} \wedge P_1^{out} \wedge l \wedge P_2^{in}$. If $pkt \neq$ `False`, then it arrives

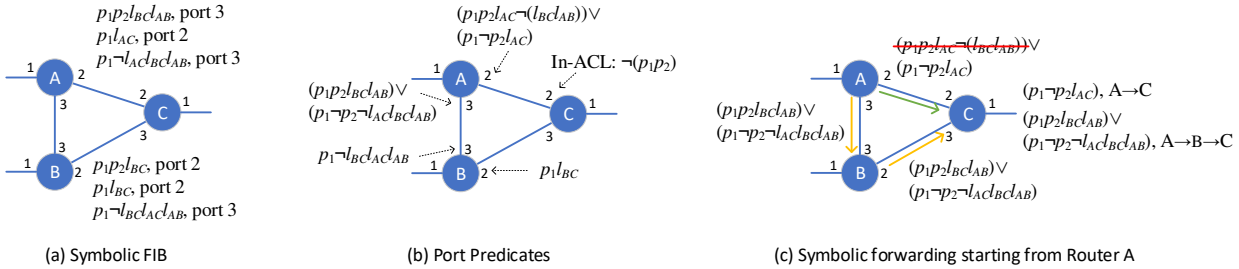


Figure 3: The process of symbolic packet forwarding.

at the port of the next-hop router. This process continues until pkt = `False`, or pkt reaches a port that is not connected to other routers.

Figure 3(c) shows the above process for the running example, where a symbolic packet is injected at router A, and reaches router C (port 1) through $A{\rightarrow}C$ and $A{\rightarrow}B{\rightarrow}C$.

Packet Failure Equivalence Class. Each symbolic packet reaching an edge port encodes the set of packet-failure tuples for a specific path, and is termed a *packet failure equivalence class (PFEC)*. Formally, we have the following definition.

Definition 1. *For a given router R, a failure scenario f, and a packet p, let $Forward_R^f(p)$ be the forwarding path of packet p starting from router R, under the failure scenario f. Two tuples $(p1, f1)$ and $(p2, f2)$ belong to the same packet failure equivalence class (PFEC) with respect to R, if and only if $Forward_R^{f1}(p1) = Forward_R^{f2}(p2)$.*

As shown in Figure 3(c), for router A, there are two PFECs, one with forwarding path $A \rightarrow C$, and the other with forwarding path $A \rightarrow B \rightarrow C$. All packet-failure tuples where packets have destination IP belonging to 128/1 and failure scenarios satisfy link AC is up belong to the first PFEC.

6 Forwarding Property Analysis

This section shows how to analyze properties based on the PFECs. We first define the properties that we consider, then give the workflow for analyzing these properties, and show how to perform three types of analyses over the properties.

6.1 Properties

We consider the following properties.

- **Reachability** $Reach(s, d, p)$: packets in p sent from s can reach d.
- **Waypointing** $Waypoint(s, d, w, p)$: packets in p sent from s can reach d, traversing waypoint w.
- **Isolation** $Isolation(s, d, p)$: packets in p sent from s can never reach d.
- **Load Balancing** $Loadbalance(s, d, p, n)$: packets in p sent from s can reach d, load balanced among n routes.

6.2 Workflow

Property analysis using SRE generally consists of three steps.

(1) Computing property BDD. First, given a property, an analyzer uses SRE to compute a *property BDD*, which is a BDD encoding all PFECs that satisfy the property (Lines 7-12 of Algorithm 2 in Appendix C). As shown in Figure 3(c), there are two PFECs that satisfy $Reach(A, C, *)$: one following $A \rightarrow C$, and the other following $A \rightarrow B \rightarrow C$. The left of Figure 4 shows the property BDD for $Reach(A, C, *)$.

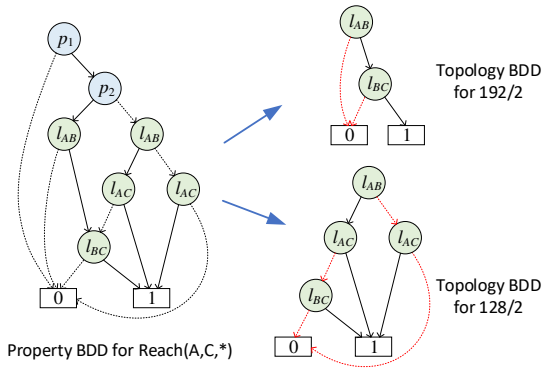


Figure 4: Analysis of reachability property.

(2) Extracting topology BDDs and packet BDDs. A property BDD can consist of multiple sets of packet headers each having a different topology condition. For example, as shown on the left of Figure 4, the solid and dashed lines starting from p_2 lead to different nodes for link variable l_{AB}. This means packets p_1p_2 (192/2) and packets $p_1\neg p_2$ (128/2) have different topology conditions. To analyze packets with different topology conditions, the analyzer uses the `Extract` function (Lines 13-18 of Algorithm 2 in Appendix C) to decouple the property BDD into a set of $(topo_i, pkt_i)$ tuples, where $topo_i$ (topology BDD) and pkt_i (packet BDD) are sub-BDDs of the property BDD, such that $\vee_i(topo_i \wedge pkt_i)$ equals the property BDD. The right of Figure 4 shows two topology BDDs for packet BDDs of 192/2 and 128/2.

(3) Analyzing topology BDDs with graph algorithms. After decoupling the property BDD into packet BDDs and topology BDDs, analyses can be performed by running graph algorithms on the topology BDDs. The analyses that SRE support include: failure tolerance analysis (§6.3), probabilistic analysis (§6.4), and differential analysis (§6.5).

6.3 Failure tolerance analysis

Definition 2. *The **link failure tolerance** for a property prop is defined as the maximum value of k satisfying that prop always holds when no more than k links fail simultaneously.*

We use $LFT(prop)$ to denote the link failure tolerance of $prop$. $LFT(prop) = 0$ means $prop$ holds when all links are up, but is violated if some single link fails; $LFT(prop) = -1$ means $prop$ does not hold even all links are up.

We show how to compute link failure tolerance for three types of properties: reachability, waypointing, and isolation.

Reachability property $Reach(s, d, *)$. Regarding reachability property, we have the following theorem.

Theorem 1. *Let $(topo, pkt)$ be a topology BDD and packet BDD tuple extracted from the property BDD of $Reach(s, d, *)$. Assign weight 0/1 to solid/dashed edges of $topo$. Then, we have: $LFT(Reach(s, d, pkt)) =$ $\texttt{ShortestPath}(topo, 0) - 1$, where $\texttt{ShortestPath}(n, 0)$ is the length of the shortest path from node n to terminal node 0.*

The proof of Theorem 1 can be found in Appendix B.

Returning to our example, the top right of Figure 4 shows that for packets in 192/2, the shortest path length from root to terminal node 0 is 1. This means that to violate the reachability, at least one link should be failed, i.e., the link failure tolerance of the reachability property is 0. On the other hand, the bottom right of Figure 4 shows that for packets in 128/2, the shortest path length is 2, meaning that the link failure tolerance of the reachability property is 1. Algorithm 2 in Appendix C summarizes the process to compute link failure tolerance for reachability properties.

Other properties. The process of computing LFT for other properties, including waypointing, isolation, etc., is almost the same. The only difference is the computation of property BDD (Line 10 of Algorithm 2 in Appendix C). For example, the property BDD for waypointing property should be the disjunction of all PFECs whose forwarding path traverse w, in addition to being sent from s to d.

6.4 Probabilistic analysis

SRE supports probabilistic analysis: given a property $prop$, computing the probability that $prop$ holds, denoted as $Prob(prop)$. In the following, we show how to compute probability for the reachability property $Reach(s, d, *)$; for other properties, the probability can be computed in a similar way.

Theorem 2. *Let $(topo, pkt)$ be a topology BDD and packet BDD tuple extracted from the property BDD of $Reach(s, d, *)$. Then, we have $Prob(Reach(s, d, pkt)) = \sum_{x \in \texttt{Truth}(topo)} Pr(x)$, where $\texttt{Truth}(n)$ is the set of all truth assignments, and $Pr(x)$ is the probability of the truth assignment x.*

Figure 4 shows that for 128/2 there are three truth assignments: $x_1 = (l_{AB} = 0, l_{AC} = 1)$, $x_2 = (l_{AB} = 1, l_{AC} = 1)$, $x_3 = (l_{AB} = 1, l_{AC} = 0, l_{BC} = 1)$. Then, $Prob(Reach(A, C, 128/2)) = \sum_{i=1}^{3} Pr(x_i)$. In the following, we show how to compute $Pr(x_i)$ for link failures and node failures.

Link failures. Assume each link fails independently with probability of $p_{down} = 0.1$ (correspondingly, $p_{up} = 0.9$), then the reachability probability is $Prob(Reach(A, C, 128/2)) = 0.1 * 0.9 + 0.9 * 0.9 + 0.9 * 0.9 * 0.1 = 0.981$. Actually, for such a failure model, we can assign weights p_{down} and p_{up} to dashed lines and solid lines, respectively in the topology BDD, and efficiently compute the probability with dynamic programming: $Prob(Reach(s, d, pkt)) = P(topo)$, $P(n) = p_{down} * P(n.d) + p_{up} * P(n.s)$, $P(1) = 1$, and $P(0) = 0$. Here, $n.d$ and $n.s$ are the two children of node n corresponding to the dashed and solid line, respectively.

Node failures (dependent link failures). When a node fails, all the links of this node will fail. This introduces dependency among link failures, and the above dynamic programming method cannot be used. Similar to [21], we use Bayesian Network (BN) to model the dependency. For this example, suppose nodes A and B fail with probability 0.01, BN will declare: $P(N_A = 0) = 0.01$, $P(N_B = 0) = 0.01$, $P(l_{AB} = 0 | N_A = 0 \vee N_B = 0) = 1$, $P(l_{AB} = 0 | N_A \neq 0 \wedge N_B \neq 0) = 0.1$, etc. Then, we can query the BN model for $P(x_i)$. Similarly, nodes or links that share the same risk can be modeled by introducing more dependency into BN, e.g., $P(N_A | N_B) = 1$.

6.5 Differential analysis

Operators are constantly changing configurations and need to know how the changes affect properties: e.g., what properties become satisfied or unsatisfied. DNA [27] can be used for such differential analyses. However, without considering link or node failures, DNA only returns "shallow differences", which may overlook undesirable differences. In the running example, suppose the operator deletes the ACL which drops packets destined for 192/2 at C. Due to the outbound route policy at C, A still chooses to route packets for prefix 192/2 towards B, and no reachability or waypointing properties are affected when all links are up. However, when links l_{AB} or l_{BC} fail, packets for prefix 192/2 will be dropped before the change but will be forwarded to C after the change. The waypointing property will be violated since packets for 192/2 will not traverse B under some link failures. In addition, the link failure tolerance changes: packets belonging to 192/2 sent by A will not reach C if l_{AB} or l_{BC} fails, before the change, while can reach C even these two links fail, after the change.

Computing differences under failures. SRE can be used to identify the above failure-triggered differences in three steps: (1) for each property, extract the topology BDD and packet BDD tuples by running steps 1 and 2 for the changed configuration. (2) for each tuple $(pkt, topo)$, where $topo$ changes from $topo'$, compute the *differential BDD*: $(topo \wedge \neg topo') \vee (\neg topo \wedge topo')$. (3) compute a truth assignment of the differential BDD. In addition, we can also compute the difference of failure tolerance and probability by computing the failure tolerance and probability for the changed configuration, and comparing to those of the original configuration.

In the running example, consider the reachability from A to C, the topology BDD of 192/2 changes, and the differential BDD encodes $l_{AC} \neg (l_{AB} l_{BC})$. One truth assignment to the differential BDD is $l_{AB} = 0$, $l_{BC} = 1$, and $l_{AC} = 1$, meaning that when link AB fails, packets 192/2 is unreachable from A to C before the change, while is reachable after the change. In addition, the tolerance increases to one after the change.

7 Optimizations

The number of possible routes under all node/link failures can explode for large networks (similar to path explosion for symbolic execution of programs). Therefore, optimizations are necessary to prune routes and make the symbolic execution tractable. Existing verifiers use different optimizations targeted at different analyses. In the following, we consider three of these optimizations, and

show how SRE can leverage them to prune a significant number of routes.

7.1 Route Pruning

Hoyan [26] observes that when considering a small number of link failures, e.g., $k \leq 3$, a lot of routes will become impossible (the topology conditions contain >3 link failures) during route computation and can be pruned. The tricky part here is that the topology condition of a route can be *partially* impossible. Suppose another router D is connected to A and B in Figure 1(a). D will receive 4 routes for prefix 128/1:

$$D{\rightarrow}A{\rightarrow}C : l_{AD}l_{AC} \quad D{\rightarrow}A{\rightarrow}B{\rightarrow}C : l_{AD}\neg l_{AC}l_{AB}l_{BC}$$
$$D{\rightarrow}B{\rightarrow}C : l_{BD}l_{BC} \quad D{\rightarrow}B{\rightarrow}A{\rightarrow}C : l_{BD}\neg l_{BC}l_{AB}l_{AC}$$

Suppose D prefers routes received from A. Then, the route $D{\rightarrow}B{\rightarrow}C$ will have a topology condition:

$$\neg(l_{AD}l_{AC}) \wedge \neg(l_{AD}\neg l_{AC}l_{AB}l_{BC}) \wedge l_{BD}l_{BC}$$
$$=(l_{BD}l_{BC}\neg l_{AD}) \vee (l_{BD}l_{BC}\neg l_{AB}\neg l_{AC})$$

If we restrict to $k \leq 1$ link failures, then only the second conjunction should be pruned. To enable the above partial pruning with SAT encoding (e.g., Hoyan), one has to represent the topology condition as a disjunction of conjunctions of link variables, in order to prune only those conjunctions with more than k negated link variables. However, due to the negations and conjunctions, the topology condition can grow very quickly, leading to what we call *topology condition explosion*, which will make the simulation time out (§8.6).

SRE realizes route pruning without topology condition explosion: since each topology condition is concisely encoded with a BDD, SRE can filter partially impossible routes by conjuncting the topology condition with a *filtering BDD* lf^k, which is a BDD representing all possible $\leq k$ link failures. For the running example with 3 links, lf^1 is constructed as:

$$lf^1 = (l_{AB}l_{AC}l_{BC}) \vee (\neg l_{AB}l_{AC}l_{BC}) \vee (l_{AB}\neg l_{AC}l_{BC}) \vee (l_{AB}l_{AC}\neg l_{BC})$$

Then, for each route with topology condition tc, SRE updates it as $tc \leftarrow tc \wedge lf^k$. The route will be pruned if $tc = False$.

Note that route pruning may under-estimate the probability that a property holds, due to ignoring all the $>k$ failure scenarios. However, when the probabilities of failures are quite small (e.g., 0.001), which is often the case [21], the probability of $>k$ failures decreases quickly with k. Therefore, if allowing for some *imprecision* (e.g., 10^{-4}), it suffices to consider only a bounded number (k) of failures and safely dropping routes with $>k$ failures. Specifically, SRE precomputes the minimum k which guarantees that the probability of $> k$ link failures is smaller than the imprecision (e.g., 10^{-4}) specified by operators, that is:

$$\sum_{m=0}^{k} \binom{n}{m} p_{down}^{m}(1-p_{down})^{n-m} > 1 - imprecision,$$

where n is the number of links, and p_{down} is the probability that a link fails.

7.2 Prefix Pruning

Config2Spec [8] observes that some properties (e.g., reachability) cannot hold under k failures due to the lack of topological connectivity. Based on this observation, Config2Spec computes (k+1)-edge-connected components (ECCs) on the topology: two nodes are in the same $(k+1)$-ECC if they remain connected when any k edges are removed. Config2Spec prunes policies (e.g., reachability) between nodes which are not in the same $(k+1)$-ECC from the set of candidate policies.

SRE leverages the observation to enable another optimization termed *prefix pruning*. Unlike Config2Spec which prunes policies to verify, SRE prunes prefixes to compute—i.e., SRE does not perform symbolic route computation for the prefixes. Before enabling this optimization, SRE first divides forwarding property analysis into several *strata*: for the $(k+1)$th stratum, SRE only considers those properties whose failure tolerance is k, thereby pruning prefixes related to those properties whose failure tolerance is $< k$. By iterating over all strata, SRE can compute failure tolerance for all properties. For the $(k+1)$th stratum, if a $(k+1)$-ECC contains only one router R, then properties related to all prefixes originated by R have failure tolerance $< k$, and those prefixes can be pruned. Moreover, since the $(k+1)$th stratum does not need to consider $> k$ link failures, SRE can apply route pruning (§7.1) to reduce the number of routes for unpruned prefixes.

Compared to route pruning which prunes routes *during* route computation, prefix pruning prunes routes *before* route computation. The joint effect of prefix pruning and route pruning is remarkable: for stratum with a smaller k, more routes will be pruned by route pruning; for stratum with a larger k, more routes will be pruned by prefix pruning. Therefore, the overall number of routes will be relatively small with the above two optimizations. As we show in our experiments, the stratified approach is faster than the one-shot approach which considers all $< k$ failures and hence does not permit prefix pruning (§8.4).

Note that prefix pruning does not affect the accuracy of failure tolerance analysis, but may under-estimate the probabilities of properties. The reason is that even the property does not hold under arbitrary k link failures, there may exist some k link failures under which the property holds, whose probabilities are not counted when prefix pruning is enabled.

7.3 Abstract Interpretation

ShapeShifter [6] applies abstract interpretation to reduce the number of routes during control plane simulation (under no failures). ShapeShifter shows that for data center networks with many redundant links and great symmetry, abstract interpretation significantly speeds up the simulation process.

SRE can apply abstract interpretation to speed up the process of SRC (§4). For example, if we only care whether there is a route to a prefix at each router, we do not need to keep the AS path and can abstract it using path length for best route selection. Then, many routes with different AS paths, but the same path length can be merged into a single route, whose topology condition is a disjunction of the topology conditions of those routes. For an 80-node fat tree with three link failures, the speedup due to this optimization is around 5× (§8.4). Unlike ShapeShifter, which concentrates on route reachability and may lose precision when there are static routes or ACLs, SRE considers packet reachability and therefore needs to preserve the next hop of each route.

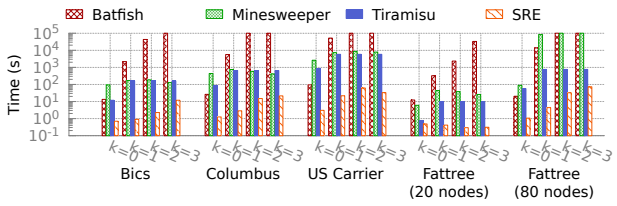


Figure 5: Time to check all-pair reachability under different number of link failures.

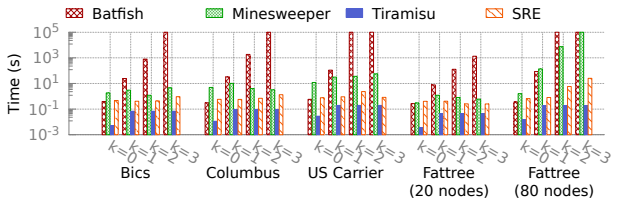


Figure 6: Time to check single-pair reachability under different number of link failures.

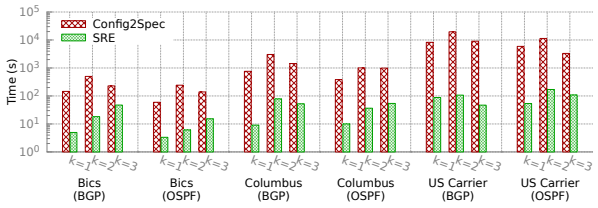


Figure 7: Running time to mine specifications.

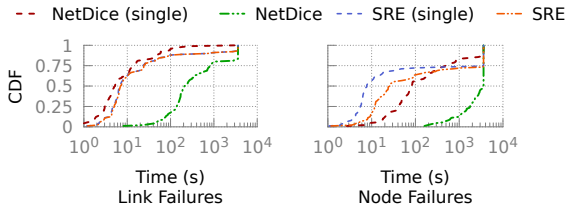


Figure 8: Running time to compute probabilities for reachability under link failures and node failures.

8 Experiments

Implementation. We implemented SRE with Java. SRE uses the JDD library [23] for BDD operations, and Batfish [1] to parse configuration files into a vendor-neutral representation. Currently, SRE supports OSPF, BGP, and static route.

Setup. All experiments run on a Linux server with two 12-core Intel Xeon CPUs @ 2.3GHz and 256G memory.

Datasets. We use three synthetic datasets and one real dataset.

(1) WAN topologies running BGP or OSPF, from Config2Spec [8]. The dataset consists of three WAN topologies (small, median, and large), consisting of 33 (48), 70 (85), and 158 (189) routers (links), respectively.
(2) WAN topologies running BGP and OSPF, from NetDice [21]. The dataset consists of 90 WAN topologies, each of which has >50 links.
(3) Fat trees running BGP or OSPF. The dataset consists of different sizes of fat trees, from 20 nodes to 245 nodes.
(4) Campus network running OSPF. The dataset consists of 67 configuration snapshots from the backbone network at a large university [19]. The network has 28 routers, 50 links, ~1K prefixes, and an average of ~75K total lines of configuration, which generate ~26K total forwarding rules. There are ~1K ACL rules.

8.1 Failure tolerance analysis

Checking reachability under failures. Figure 5 shows the running time of SRE and three other configuration verifiers to check all-pair reachability on the three WAN topologies and the fat tree topologies (20 nodes and 80 nodes). For the WAN topologies, SRE is generally >10× faster than the other verifiers. For the fat tree topologies, SRE is >100× faster than Batfish and Minesweeper, and faster than Tiramisu. We also include the results for checking single-pair reachability, shown in Figure 6. We can see that SRE is faster than or comparable to Batfish and Minesweeper, but slower than Tiramisu. This indicates that SRE is a better choice for reasoning about a large header space and failure space, but not optimized for reasoning about a specific prefix or failure.

Specification mining. We use SRE to mine policies from configurations. To compare with Config2Spec [8], we use the three WAN topologies, and consider four types of policies—reachability, waypoint, isolation, and load balancing. Figure 7 shows SRE mostly takes <100 seconds to mine all the policies, 1-2 orders of magnitude faster than Config2Spec.

8.2 Probabilistic analysis

We run SRE on the 90 WAN topologies [2] to compute probabilities for reachability and waypointing properties with both link and node failures. For each prefix, we consider the reachability from each router to the prefix, and select a random waypoint. We set the probabilities of node failure and link failure to 0.0001 and 0.001, respectively (the same as NetDice). Both SRE and NetDice return the same probabilities for reachability and waypointing properties, within an imprecision of 10^{-4} on all topologies. As shown in Figure 8, for link failures, NetDice is faster than SRE when computing the probability of a single reachability, but SRE is 1-2 orders of magnitude faster than NetDice when computing the probabilities of all reachabilities, except some large topologies (up to 2320 edges) for which both SRE and NetDice time out after 1 hour. For node failures, NetDice can compute probabilities of some reachabilities that SRE cannot compute, while SRE is >2 orders of magnitude faster than NetDice when computing probabilities of all reachabilities, except the large topologies. This shows the advantage of SRE in reasoning about the product space of packets and failures. The results for waypointing probability (Appendix D) are similar.

8.3 Differential Analysis

We use SRE to compute the differences after a configuration change. We consider the 10 atomic changes synthesized by DNA [27] and

Dataset	No. Routes	Reduction Ratio		
		RoutePrune	+PrefixPrune	+Abstract
Bics	3,819,240	98.32%	91.80%	61.42%
Columbus	25,382,778	98.81%	95.76%	59.09%
US Carrier	280,624,242	98.55%	99.20%	74.55%
Fattree(20)	146,040	97.55%	100.00%	0.00%
Fattree(80)	BDD limit	(379,552)	0.00%	93.82%
Fattree(125)	BDD limit	(2,389,050)	0.00%	96.97%

Table 2: The reduction in routes when applying different optimizations ($k = 3$, BGP). When the BDD node count limit is reached, the corresponding number in the third column is the number of routes.

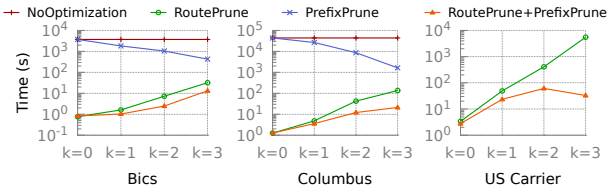


Figure 9: Time to compute link failure tolerance of reachability, with and without route/prefix pruning.

apply these changes on the Bics WAN topology. For each change, we run SRE with $k = 0$ to compute the differences DNA will find, and we run SRE with $k = 3$ to get the differences under failures. DNA can detect differences for 5/10 of the updates, while SRE can detect differences in failure tolerance and probability for 7/10 and 10/10 of the updates, respectively. This means SRE can be used to find differences that only manifest under specific failures.

8.4 The effectiveness of optimizations

We now quantify the effectiveness of the three optimizations (§7).

WAN topologies. Figure 9 shows the running time of SRE when computing failure tolerance with and without route pruning and prefix pruning (abstract interpretation is not quite effective, and is not shown here). We can see that:

(1) optimizations are quite necessary for SRE to scale. For US Carrier, without route pruning, the number of required BDD nodes exceeds the limit supported by the JDD library [23] (see §8.5 for details), while with route pruning and prefix pruning, the running time is within 100 seconds. The scalability comes from the significant reduction of routes (Table 2).

(2) different optimizations have different effectiveness for different number of failures (k). Route pruning is more effective for smaller k, when a lot of routes have $> k$ failures, while prefix pruning is more effective for larger k, when a lot of prefixes whose related properties cannot tolerate $\geq k$ failures.

(3) stratification approach performs better than one-shot approach, i.e., computing failure tolerance in a single round considering all $1, 2, \ldots, k$ failures, and cannot enable prefix filtering. For example, for US Carrier $k = 3$, the one-shot approach uses 5500 seconds (the RoutePrune time for $k = 3$), while the stratification approach uses 120 seconds (sum of the RoutePrune+PrefixPrune time for $k = 0, 1, 2, 3$).

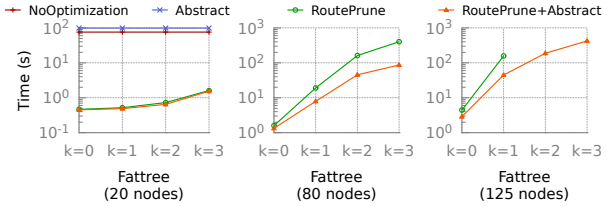


Figure 10: Time to compute link failure tolerance of reachability, with and without abstract interpretation.

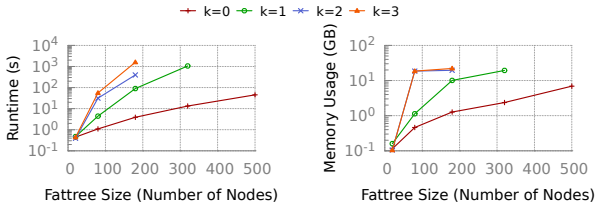


Figure 11: Running time and memory usage when checking all-pair reachability for different sizes of fat trees.

Fat trees. Figure 10 shows the running time of SRE when computing link failure tolerance with and without abstract interpretation and route pruning (prefix pruning is not effective, except 20 nodes and $k = 3$, and thus not shown here). As can be seen, abstract interpretation becomes more effective for larger fat trees with more redundant links, when more routes with equal path length can be merged. Without abstract interpretation, the number of BDD nodes required for the 125-node fat tree for $k = 2, 3$ exceeds JDD's limit. Similar to the WAN topologies, route pruning is quite effective.

8.5 Scalability

To evaluate whether SRE can scale to even larger networks with >1000 links, we use SRE to analyze the failure tolerance of all-pair reachability on different sizes of fat trees. Figure 11 shows the running time and peak memory usage of SRE for different number of link failures. As shown in Figure 11(a), for fat tree with 320 nodes (2048 links), SRE finishes when there are at most one link failure, while for fat trees with 500 nodes and 4000 links, SRE finishes only when there are no failures.

This is due to the limitation of node table size in JDD, which uses an array of integers to store all BDD nodes. Since each BDD node uses three integers, the maximum (theoretical) number of nodes is $(2^{31} - 1)/3$, which is roughly 7.16×10^8. Since JDD uses 22 Bytes for each node, it consumes approximately 16GB for maintaining BDD nodes. To confirm this, we allocate 100GB to Java Virtual Machine, and observe that the peak memory usage is bounded by around 20GB, which is comparable to 16GB. Therefore, we expect SRE can scale to larger fat trees with another BDD library that can hold more nodes.

8.6 SAT or BDD?

In this experiment, we replace the encoding of topology condition with SAT formula, similar to Hoyan [26], and show how it compares to SRE which uses BDDs. We randomly select 10 prefixes from

Dataset		k=0	k=1	k=2	k=3
Bics	TC Length	480	2,116	8,195	28,651
	Time (s)	0.96	1.23	2.48	13.13
	Timeout	0/10	0/10	0/10	0/10
Columbus	TC Length	1470	16,726	147,009	813,122
	Time (s)	1.37	3.25	91.27	1435.49
	Timeout	0/10	0/10	0/10	3/10
US Carrier	TC Length	4,930	79,030	809,318	-
	Time (s)	2.80	15.96	712.38	-
	Timeout	0/10	0/10	1/10	10/10

Table 3: Length of topology condition and running time with SAT encoding.

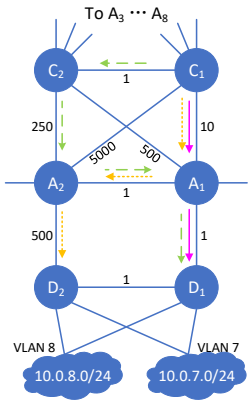


Figure 12: The topology of the campus network and the three packet forwarding paths (in different colors) from a core router C_1 to 10.0.7.0/24. The number along each link denotes the OSPF cost of that link.

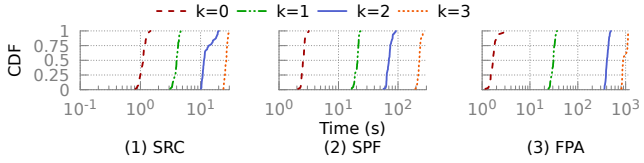


Figure 13: Running time of SRE on campus network.

the three WAN topologies, and run symbolic route computation. Table 3 shows that with increasing network size and value of k, more prefixes will time out, e.g., for US Carrier $k = 3$, all the 10 prefixes time out. The reason is topology condition explosion: the formula length encoded with SAT grows quite fast when k increases, making the updating of the topology condition extremely slow.

8.7 Real Network

We use SRE to check reachability in the campus backbone network. As shown in Figure 12, the campus backbone network has a hierarchical structure, with 2 core routers (C_1 and C_2), 8 aggregation routers (A_1–A_8), and 18 distribution routers (D_1–D_{18}). The aggregation and distribution routers are deployed in primary-backup pairs (e.g., A_1 and A_2). Each access VLAN (e.g., VLAN 7 associated with subnet 10.0.7.0/24) is connected to a pair of distribution routers.

First, we check all-pair reachability between all access VLANs. Figure 13 shows the running time for the two stages of SRE (SRC and SPF) and property analysis (FPA). The distribution is over the 67 configuration snapshots. SRE generates the same FIBs as Batfish when there are no failures ($k = 0$). We can see that for this campus network, SRE takes around 1000 seconds. We also run Minesweeper and Tiramisu, both of which cannot run to completion. This is due to the existence of ~1K ACL rules, ~1K prefixes, >1K VLANs, and multiple VRFs.

Second, we compute the failure tolerance for reachability from each core router to each access VLAN: e.g., Reach(C_1, 10.0.7.0/24), Reach(C_2, 10.0.7.0/24), etc. The failure tolerance computed by SRE and Minesweeper are both 1—i.e., an access VLAN is always reachable from C_1 or C_2 if there are ≤ 1 link failures, but unreachable if there are ≥ 2 link failures such as when $l_{A_1D_1}$ and $l_{A_2D_2}$ fail simultaneously.

9 Limitations

No performance gains when analyzing a single point in the header or failures space. SRE is aimed at scaling to the product space of packets and failures, and therefore not optimized for checking a single prefix under failures. As shown in §8, SRE is slower than Tiramisu for checking single-pair reachability, and comparable with Batfish and Minesweeper when there are no failures (Figure 6).

No support for cross-path/cross-flow properties. SRE currently does not support properties that require reasoning about multiple forwarding paths of the same flow [11] or the forwarding behaviors of multiple flows [21].

No incremental computation. SRE currently does not support incremental computation [27]. When configurations change, SRE needs to re-run symbolic route computation and symbolic packet forwarding, and re-check the properties.

10 Conclusion

Symbolic Router Execution (SRE) is a general and scalable network verification engine that supports various types of analyses. SRE symbolically executes the network model to discover packet failure equivalence classes (PFECs) to scale to the product space of headers and failures. By encoding symbolic headers and failures with BDDs, SRE enables operators to analyze properties with graph algorithms on BDDs, agnostic of failure models or network semantics. Our future work includes overcoming the limitations of SRE, and experimenting with other BDD libraries.

Acknowledgement. We thank our Shepherd Ennan Zhai, and all the anonymous SIGCOMM reviewers for their valuable comments and suggestions. This work is partially supported by the United States National Science Foundation (No. 1763512).

Ethical issues. This work does not raise any ethical issues.

References

[1] [n. d.]. Batfish. https://github.com/batfish/batfish.
[2] [n. d.]. NetDice. https://github.com/nsg-ethz/netdice.
[3] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast and General Network Verification. In *USENIX NSDI*.
[4] Henrik Reif Andersen. 1997. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* (1997).
[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *ACM SIGCOMM*.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. In *ACM POPL*.

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. 2009. Mining policies from enterprise network configuration. In *ACM IMC*.

[8] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *USENIX NSDI*.

[9] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* 100, 8 (1986), 677–691.

[10] 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 *USENIX OSDI*.

[11] 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 *USENIX NSDI*.

[12] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*.

[13] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*.

[14] 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 International Conference on Programming Language Design and Implementation (PLDI)*.

[15] Nick Giannarakis, Alexandra Silva, and David Walker. 2021. ProbNV: probabilistic verification of network control planes. *Proc. ACM Program. Lang.* 5, ICFP (2021).

[16] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *USENIX NSDI*.

[17] Ali Kheradmand. 2020. Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns. In *ACM Symposium on SDN Research*.

[18] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[19] David Plonka and Andres Jaan Tack. 2009. An Analysis of Network Configuration Artifacts. In *Proceedings of the 23rd Large Installation System Administration Conference*.

[20] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*.

[21] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *ACM SIGCOMM*.

[22] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM*.

[23] Arash Vahidi. [n. d.]. JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/.

[24] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM OOPSLA*.

[25] Hongkun Yang and Simon S Lam. 2013. Real-time verification of network properties using Atomic Predicates. In *IEEE ICNP*.

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

[27] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential Network Analysis. In *USENIX NSDI*.

A Algorithms for Symbolic Route Computation

Algorithm 1: `UpdateRIB`(R)

Input: R: the router whose RIB is to be updated.

```
1  lists ← {};
2  routesOut ← {};
3  Sort(routesIn);
4  foreach route ∈ routesIn do
5      list ← rib.Get(route.prefix);
6      if list.contains(route) then
7          r ← list.getRoute(route);
8          r.tc^in ← route.tc^in;
9      else
10         list.InsertRoute(route);
11     index ← list.getIndex(route);
12     if index < list.changePos then
13         list.changePos ← index;
14     lists ← lists ∪ {list};
15 foreach list ∈ lists do
16     matched ← False;
17     foreach route ∈ list[0 : list.changePos] do
18         matched ← matched ∨ route.tc^in;
19     foreach route ∈ list[list.changePos : list.len] do
20         tc ← ¬matched ∧ route.tc^in;
21         if tc ≠ route.tc^rib then
22             route.tc^rib ← tc;
23             routesOut ← routesOut ∪ {route}
24         matched ← matched ∨ route.tc^in;
25 foreach route ∈ routesOut do
26     foreach N ∈ Neighbors do
27         if PolicyAllow(R, N, route) then
28             r ← route;
29             r.tc^in ← route.tc^rib ∧ Link(R, N);
30             r.tc^rib ← False;
31             Advertise(N, r);
```

B Proof of Theorem 1

Proof. Let T be the link failure tolerance, and L be the length of shortest path. We will prove $T = L - 1$ by showing (1) $T < L$, and (2) $T \geq L - 1$. First, since there is a path from root to `False` whose length is L, then there exists a topology condition where L links are down, such that the reachability does not hold. That is, we have the failure tolerance $T < L$. Second, suppose $T < L - 1$, then we have a topology condition where $(L - 1)$ links are down and all other $(N - L + 1)$ links are up, such that the reachability does not hold. The condition corresponds to a path to `False` which has at most $(L - 1)$ dashed edges, since the $(N - L + 1)$ links either correspond to solid edges, or do not appear on the path. This contradicts the fact that the shortest path to `False` is L. □

C Algorithms for Failure Tolerance Computation

Algorithm 2: LFTReach(src, dst, hdr, $\mathcal{P}$)

Input: src: the source; dst: the destination; hdr: the header specification; $\mathcal{P}$: the set of all PFECs.
Output: LFT: a set of tuples (src, dst, pkt, k), meaning the link failure tolerance for $Reach(src, dst, pkt)$ is k.

$LFT \leftarrow \{\}$;
$reach \leftarrow$ GetPropertyBDDReach(src, dst, hdr);
$extracted \leftarrow$ Extract($reach$, *wildcards*);
foreach $(topo, pkt) \in extracted$ **do**
  $k \leftarrow$ ShortestPath$(topo, 0) - 1$;
  $LFT \leftarrow LFT \cup \{(src, dst, pkt, k)\}$;
Function GetPropertyBDDReach(src, dst, hdr):
  $reach \leftarrow$ False;
  foreach $p \in \mathcal{P}$ **do**
    if $p.src = src$ *and* $p.dst = dst$ **then**
      $reach \leftarrow reach \vee p$;
  return $reach \wedge hdr$;
Function Extract($node, pkt$):
  if $node \in \{$True, False$\}$ *or* $var(node) \in Links$ **then**
    return $\{node, pkt\}$;
  $p_l \leftarrow pkt, p_l[var(node)] \leftarrow 0$;
  $p_r \leftarrow pkt, p_r[var(node)] \leftarrow 1$;
  return Extract$(node.l, p_l) \cup$ Extract$(node.r, p_r)$;

D Extra Experiment Results

Figure 14 shows running time to compute probabilities for waypointing property under link failures and node failures.

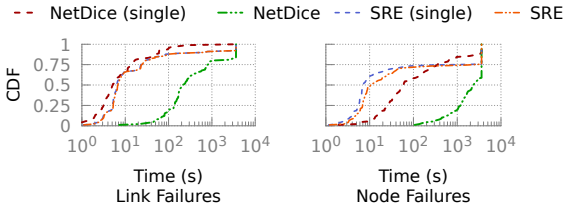


Figure 14: Running time to compute probabilities for waypointing property under link failures and node failures.