Diagnosing and Repairing Distributed Routing Configurations Using Selective Symbolic Simulation

Rulan Yang [1], *Gao Han* [1], *Hanyang Shao* [1], *Xiaoqiang Zheng* [1], *Xing Fang* [1], *Ziyi Wang* [1], *Lizhao You* [1,*], *Ruiting Zhou* [2], *Linghe Kong* [3], *Ennan Zhai* [4], *Qiao Xiang* [1,*], *Jiwu Shu* [1,5]
[1]*Xiamen University*, [2]*Southeast University*, [3]*Shanghai Jiao Tong University*, [4]*Alibaba Cloud*, [5]*Minjiang University*

Abstract

Although substantial progress has been made in automatically verifying whether distributed routing configurations comply with certain intents, diagnosing and repairing configuration errors remains manual and time-consuming. To fill this gap, we propose S^2Sim, a novel system for automatic routing configuration diagnosis and repair. Our key insight is that by deriving a set of contracts that guarantees an intent-compliant variant of the erroneous configuration, we can systematically check for all contract violations in the configuration via symbolic simulation to pinpoint and repair the errors. S^2Sim also introduces a series of extensions to support complex configurations (*e.g.*, ACL, route aggregation and multi-path routing), networks (*e.g.*, underlay and overlay networks), and intents (*e.g.*, k-link failure tolerance). We fully implement S^2Sim and evaluate its performance using real configurations from two major providers and synthesized configurations composed from their real errors and real-world topologies with different scales $O(10)$ to $O(1000)$. Results show that S^2Sim accurately and efficiently diagnoses and repairs real configuration errors (*i.e.*, up to 20 seconds in real networks of $O(100)$ nodes and up to 15 minutes in synthesized networks of $O(1000)$ nodes).

1 Introduction

Many network configuration verification tools, also called control plane verification (CPV) tools, have been designed and deployed [2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 24, 25, 28, 35, 40, 43, 44, 45] . They analyze the configurations of routers and switches to determine whether these configurations, when deployed in production networks, would yield a data plane that satisfied a series of pre-specified intents (*e.g.*, reachability, blackhole freeness, waypointing, and loop freeness).

Diagnosing and repairing configuration errors remains manual, time-consuming and error-prone. Despite the substantial progress in expanding the capability of CPV and accelerating them, once a set of configurations are deemed erroneous, it is still up to network operators to manually locate the errors and repair them. Although some CPV tools (*e.g.*, Minesweeper [3]) also return a counter example to users, but repairing the configurations, in the worst case, requires enumerating all possible counter examples, which is NP-complete.

Some studies have attempted to tackle the problems of diagnosing and repairing network configurations. Earlier data provenance-based tools [8, 27, 36, 37, 46] analyze the causal relationship among network events to find the root causes (*e.g.*, link failures) of network behaviors (*e.g.*, updates of forwarding rules). However, they require re-implementing routing protocols and their configurations in a declarative datalog dialect (*e.g.*, NDlog [46]), which brings additional learning curves for operators and limits their deployment. They also could not identify latent errors (*i.e.*, errors that have not caused a failure yet, but will do once other configurations are updated.)

Recent tools based on program analysis cannot accurately localize and repair configuration errors (§2). Recent tools [14, 15, 26] resort to program analysis to diagnose or repair configuration errors, but have different limitations. CEL [15] extends Minesweeper to model network configurations and intents into an SMT formula, and computes its minimal correction set (*i.e.*, a subset of constraints whose removal would allow the remaining of the formula to become satisfied) as the configuration errors. However, it cannot support AS-path related configurations (*e.g.*, a route filter matching on ASes in the AS path) due to the path encoding explosion of the SMT formula. CPR [14] models network configurations using an abstract representation and repairs them using constraint programming, but it cannot support local preference-related configurations due to its graph-based abstraction. ACR [26] applies a spectrum-based method to rank potential erroneous lines and uses an experience-based method to find a possible repair. Not only could it require multiple trial-and-error without success, it also relies on test coverage based tools (*e.g.*, NetCov [39]) to first identify some lines as candidates, which could miss real errors even in a simple network.

In this paper, we systematically investigate the important

[*]Lizhao You and Qiao Xiang are co-corresponding authors.

problem of how to accurately and efficiently diagnose and repair errors in network configurations.

Proposal: Explore variants of the original erroneous configuration to find an intent-compliant one, and compare their differences for error diagnosis and repair. Instead of analyzing the given configuration to find out what went wrong, we resort to a different mindset: if we can find an intent-compliant configuration that is similar to the original erroneous one, their differences essentially suggest the error and the repair. At first glance, this appears to be a *catch-22*: in order to design a tool to diagnose and repair configurations, we must first have a tool that can find an intent-compliant variant of the original configuration (*i.e.*, updating small portion of the original one to make it intent-compliant).

Key insight: Find a set of contracts that guarantee an intent-compliant variant, and detect where they were breached in the original configuration as errors via symbolic simulation (§3). We circumvent the paradox above by not searching for a concrete configuration, but a set of *contracts* such that if a configuration satisfy them all, it must be intent-compliant. Informally, a contract is a Boolean predicate that specifies the behavior of a router (*i.e.*, whether to peer with another router and whether to discard/prefer/announce a route). By simulating the erroneous configuration symbolically to identify all the snippets that could violate the intent-compliant contracts, we can localize and repair the errors in the configuration. While a previous position paper by Yang et al. [42] also suggests the promise of diagnosing and repairing network configuration using symbolic simulation, it fell short in answering several important questions, including (1) how to support complex configurations (*e.g.*, access control lists, route aggregation, and multi-path routing), (2) how to cope with complex networks where multiple routing protocols (*e.g.*, path-vector and link-state) co-exist, and (3) how to diagnose and repair the fault-tolerance of configuration. To this end, we design S^2*Sim*, a generic, efficient, lightweight tool for network configuration diagnosis and repair with three key designs (D1-D3).

D1: Derive intent-compliant contracts from erroneous data plane and intents, and check their violations with selective symbolic simulation (§4). After finding a configuration is erroneous via simulation, S^2*Sim* designs an algorithm based on DFA-multiplication to compute an intent-compliant data plane with a small difference from the erroneous one. It derives a set of contracts that is sufficient and necessary to yield this data plane. Next, S^2*Sim* starts another simulation of the erroneous configuration, which is selective and symbolic. At each step, if a contract is violated, S^2*Sim* forces the simulation to obey this contract and switch the simulation to a symbolic configuration variant. Because the simulation obeys all the derived contracts, it eventually yields the intent-compliant data plane. As such, S^2*Sim* can localize the errors in the original configuration by mapping the violated contracts to erroneous configuration snippets and repair them using an algorithm based on constraint programming. S^2*Sim* also extends to support a wide range of complex configurations such as route aggregation, access control lists (ACL), and multi-path routing.

D2: Use an assume-guarantee approach to diagnosis and repair a network with multiple routing protocols (§5). For networks running different routing protocols in underlay and overlay (*e.g.*, OSPF as underlay and BGP as overlay), S^2*Sim* diagnosis and repair layers separately for modularity. It first assumes the proper functioning of the underlay network (*e.g.*, reachability between overlay-neighboring routers) and diagnoses and repairs the overlay network to be intent-compliant. It then uses the assumption as the intents for the underlay network and repeats the diagnosis and repair process.

D3: Derive fault-tolerant contracts for configuration fault-tolerance (§6). To diagnose and repair configuration so that it is intent-compliant even when there are up to k link failures in the network, S^2*Sim* extends the basic design to compute an intent-compliant data plane that is also fault-tolerant, and derives fault-tolerant contracts accordingly. It then incorporates the selective symbolic simulation to check for contract violations under fault-tolerant contracts and use these violations for error localization and repair.

Evaluation with production data (§7). We implement a closed-source version of S^2*Sim* for a major provider as a plug-in of its in-house simulation CPV tool, and an open-source prototype[1] as a plugin of Batfish [13]. We evaluate S^2*Sim* with (1) functionality demos on a small network [29]; (2) experiments using real configurations from production networks of two major providers; and (3) experiments using configurations synthesized from the providers' real errors and real-world topologies of different scales $O(10)$ to $O(1000)$. Results show that S^2*Sim* accurately and efficiently diagnoses and repairs configuration errors in real networks (*i.e.*, up to 20 seconds in DC-WAN and IPRANs with $O(100)$ nodes) and synthesized large-scale networks (*i.e.*, up to 15 minutes in IPRANs and DCNs with $O(1000)$ nodes).

2 Why Existing Tools Do Not Work

In this section, we demonstrate the limitations of related tools through an experiment in a simple network. The screenshots of outputs of all our experiments are in Appendix A.

An example network. Consider a small network of six routers running BGP in Figure 1. For simplicity, we use the ID of routers as their AS numbers. The destination IP prefix p is at router D. The intents of the operator are: (1) all routers can reach prefix p; (2) router A must waypoint router C; and (3) router F must avoid router B. All the routers use the default BGP configuration except for the snippets of router C and F shown in the figure, where C does not send route with prefix p to B, and F sets a higher local preference for all routes whose AS-path contains C. Because it is such a small network, we

[1] https://github.com/sngroup-xmu/S2Sim

C's configuration snippet

```
ip prefix-list pl1 seq 5 permit p
!
route-map filter deny 10
  match ip address prefix-list pl1
!
route-map filter permit 20
!
bgp C
 neighbor B route-map filter out
!
```

Intents:
(1) All routers can reach p
(2) A must waypoint C
(3) F must avoid B

F's configuration snippet

```
ip as-path al1 permit _C_
!
route-map setLP permit 10
 match as-path al1
 set local-preference 200
!
route-map setLP permit 20
 set local-preference 80
!
bgp F
 neighbor A route-map setLP in
 neighbor E route-map setLP in
!
```

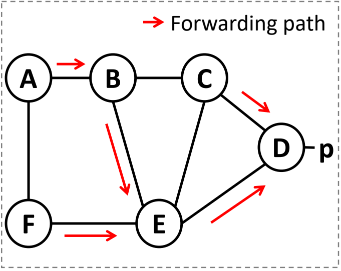


Figure 1: The example network.

can simulate its execution on paper and observe that while all the routers can reach p, the path of A does not satisfy intent 2.

Errors in the configuration: the ground truth. There are two errors in the configuration. One may quickly observe that one is the export policy at C that denies routes with prefix p to B. However, removing it only partially fixes the configuration. For example, if C removes this policy, B will receive and prefer $[B,C,D]$ over $[B,E,D]$ as its path to reach p because C has a lower ID than E, and A will also use path $[A,B,C,D]$ to reach p. But F will also switch to $[F,A,B,C,D]$ because it has a policy to prefer the path that contains AS C over all other paths. Therefore, router F's policy to prefer the AS-path that contains C is also an error and needs to be removed.

Outputs of representative configuration verification/diagnosis/repair tools. We select five recent tools to examine the configuration of this small network against the three intents. To make sure the experiment is fair, we use the open-source prototypes from the authors. We find that although some of them can determine that the configuration is incorrect, none of them can correctly locate and repair the two errors.

Batfish [7, 13] correctly determines the configuration is erroneous but cannot locate the errors. As a simulation-based CPV, Batfish computes the paths of each router based on their configurations and alerts us that these paths violate intent 2 (Figure 13 in Appendix A). However, it neither identifies the incorrect configuration nor suggests a fix.

Minesweeper [3] also correctly determines the configuration is erroneous and cannot locate the errors. As an SMT-based CPV, Minesweeper alerts us that the configuration is erroneous and provides a corresponding counter example (Figure 14 in Appendix A). Likewise, it provides no suggestions for localizing or repairing the configuration error.

CEL [15] fails to find all errors in the configuration. CEL analyzes the SMT formula constructed by Minesweeper and computes its minimal correction set (MCS) as the errors. However, CEL only finds the errors at router C in this example. This is because CEL's encoding is based on Minesweeper,

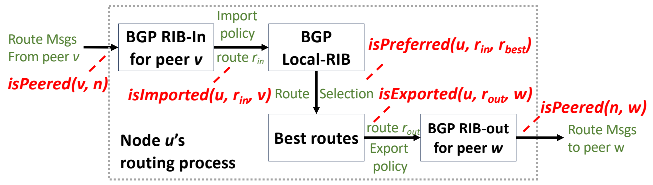


Figure 2: The BGP process of a router. The configuration determines the routing contracts (shown in red).

which does not support AS-path related configuration because of the path explosion. (Figure 15 in Appendix A).

CPR [14] fails to repair any error in this configuration. CPR uses an abstract graph representation to model the network configuration and finds the repair using constraint programming. CPR identified the configuration as erroneous, but incorrectly repaired the error by adding an ACL in router A to block all traffic from A to D (Figure 16 in Appendix A). This is because CPR's graph model incorrectly determines that A can reach C and cannot support the local preference-related configurations.

ACR [26] cannot locate or repair any error. ACR adopts a trial-and-error approach. In each iteration, it applies a spectrum-based method to rank potential erroneous lines, tries to repair them using an experience-based method, and validates the repair using CPV (*e.g.*, Batfish). Although it does not have an open-source prototype, it suggests that it uses a configuration test coverage tool (*i.e.*, NetCov [39]) to first identify configuration lines related to the errors as candidates. Therefore, we implement ACR by ourselves with NetCov as a key component. In this example, however, when we send NetCov a test case of path from A to D, the candidate lines NetCov returns do not include any route policies at C (Figure 17 in Appendix A), the actual errors in the network. This is because NetCov leverages positive provenance to identify the related configuration for existing routes, but it cannot locate the configuration contributing to the nonexistence of a route. As a result, ACR cannot locate or repair the configuration after repeating the trial-and-error process multiple times.

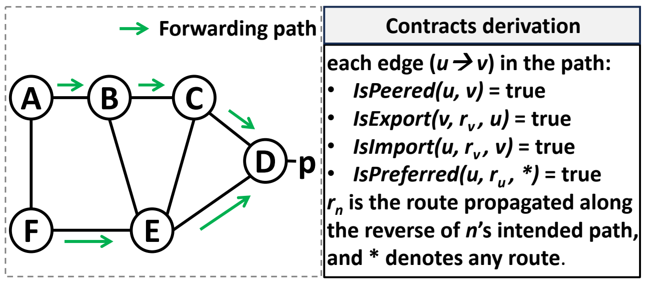
Takeaway: an accurate tool for configuration diagnosis and repair is still missing. Representative existing tools have different limitations, making it difficult for them to locate or repair the configuration errors, even in a simple network with simple configurations.

3 S^2Sim in a Nutshell

We use the same example in Figure 1 to illustrate the key insight and basic workflow of how S^2Sim accurately diagnoses and repairs network configuration. We also provide an interactive demo [29] of this example to demonstrate the functionality of our tool.

3.1 Contracts

A contract is a predicate that specifies the behavior of a router. To be concrete, a routing process runs as an event-driven program. It receives route updates (*e.g.*, BGP and link-state



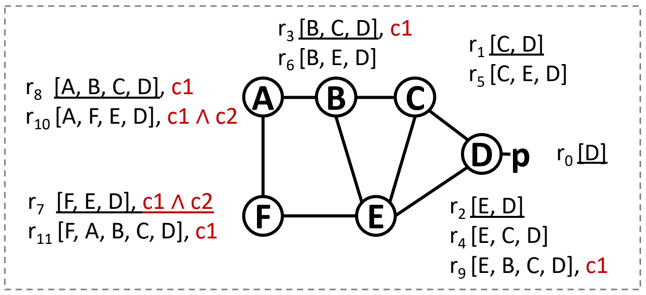
Contracts derivation

each edge ($u \rightarrow v$) in the path:

- *IsPeered(u, v)* = true
- *IsExport(v, r_v, u)* = true
- *IsImport(u, r_v, v)* = true
- *IsPreferred(u, r_u, *)* = true

r_n is the route propagated along the reverse of n's intended path, and * denotes any route.

Figure 3: The intent-compliant data plane and contracts of the example network.



Violated contracts	
c1: isExported (C, r_1, B) == True	**c2:** isPreferred (F,r_7,r_{11}) == True

Figure 4: The symbolic simulation of the example network.

route announcement) and processes them through a series of conditional statements. Each Boolean condition in this program is a *contract*. For example, Figure 2 provides a basic overview of a BGP process, comprising four types of contracts: *isPeered*, *isImported*, *isPreferred*, and *isExported*, each governing specific routing behavior within the process. Table 1 provides a summary of common contracts.

Key observation: network configuration decides the values of contracts. Given a network configuration script, it does not change the structure of the event-driven program, but only the value of its contracts. Taking the route policy of router C in Figure 1 as an example, this contract is to set the *isExported* value to be *false* for all routes with prefix p to router B.

3.2 Workflow

The basic workflow of S^2*Sim* consists of three steps: (1) find a set of contracts that ensure an intent-compliant configuration; (2) simulate the original configuration symbolically and selectively to find all violations of these contracts; (3) locate errors by mapping violated contracts to configuration snippets and compute a conflict-free repair using constraint programming. We illustrate this procedure using the example in Figure 1.

Step 1: Derive intent-compliant contracts from the intents and erroneous data plane. Motivated by recent network synthesis (*e.g.*, Propane [6] and Contra [19]) and verification tools (*e.g.*, RCDC [20] and Tulkun [38]), we derive intent-compliant contracts from an intent-compliant data plane. A strawman to find such a data plane is to express intents in a regular expression and multiply it with the topology (*i.e.*, take their cross product). Although it works, the resulting data plane could be very different from the one yielded by the original erroneous configuration. For example, paths $[A,F,E,C,D]$ and $[A,B,C,D]$ are both compliant with $A's$ intent. However, if we derive contracts from the former one and use them to guide the diagnosis and repair, it could result in a large number of changes of lines in the configuration (including changing A and E's configurations).

With this in mind, we reuse the intent-compliant part of the erroneous data plane from the original configuration as a starting point, compute an intent-compliant data plane with a slight difference from the erroneous one, and use it to derive intent-compliant contracts. Our intuition is that the smaller the change in the data plane, the fewer contract violations in the configuration, and the fewer lines in the configuration need to be updated. Specifically, we begin with the set of satisfied paths in the original erroneous data plane as our initial path constraints. For each unsatisfied intent, we compute the shortest valid path that satisfies both its regular expression and the current path constraints, and add it to the path constraints. During this iteration, if no valid path exists for an intent, we backtrack by gradually relaxing path constraints, *i.e.*, removing one path at a time from path constraints and marking the corresponding intents as unsatisfied, until a valid path for the current intent is found, then proceed to the next intent. To address the combinatorial complexity introduced by the ordering of path finding and backtracking operations, we introduce two principles in §4.1, which ensure that we can efficiently construct an intent-compliant data plane while reusing as many segments of the original data plane as possible.

In the example in Figure 1, router B, C, E and F each has an intent-compliant path, but A does not. We try to find a valid path for A's intent (*i.e.*, $A.^*C.^*D$ and loop-free) and uses the paths of other routers as guidance constraints. However, no valid path exists for A's intent. Therefore, we remove a satisfied path $[B,E,D]$ from the path constraints and mark B's intent as unsatisfied, this relaxation allows us to find a valid path for A ($[A,B,C,D]$). In the next iteration, we reprocess B's intent and find $[B,C,D]$ is already a valid path. In the end, we get an intent-compliant data plane in Figure 3 with only one link different from the erroneous data plane.

From an intent-compliant data plane to intent-compliant contracts. The reverse of the data plane computed above (*i.e.*, changing the direction of all links) provides a set of *sufficient and necessary* conditions for the routers to yield it. We call them the *intent-compliant contracts*. To be concrete, a forwarding path $[\dots,R_{i-1},R_i,R_{i+1},\dots]$ will exist in the data plane if and only if for each router R_i, it peers with R_{i+1}, imports the route $[R_{i+1},\dots]$, prefers it over other routes (*i.e.*, the best route), and exports it to R_{i-1}. In other words, when R_i processes the route $[R_{i+1},\dots]$, all its contracts (*i.e.*, *isPeered*, *isImported*, *isPreferred*, and *isExported*) must be *true*. We call such routes intent-compliant routes. Figure 3 lists the

Contracts	Descriptions	Mapped Configuration Snippets
isPeered (u, v)	Whether to establish a peer session between nodes u and v (for path-vector protocols)	Peer snippets on u and v
isEnabled (u, v)	Whether the interfaces between nodes u and v are in the same area (for link-state protocols)	Interface snippets on u and v
isPreferred (u, r, r')	Whether to prefer route r over r' at node u	Import policy snippets that match r and r' on u
		Link cost snippets on nodes along paths of r and r'
isExported (u, r, v)	Whether to export a route r to v at node u	Export policy snippets on u matching route r for v
isImported (u, r, v)	Whether to import a route r from v at node u	Import policy snippets on u matching route r from v
isEqPreferred (u, r, r')	Whether to equally prefer routes r and r' at node u	Import policy snippets matching r and r' and multi-path policy on u
isForwardedIn (u, p, v)	Whether to forward packet p from v at node u	Inbound ACL snippets on u that match packet p from v
isForwardedOut (u, p, v)	Whether to forward packet p to v at node u	Outbound ACL snippets on u that match packet p to v

Table 1: Common contracts and their mapped configuration snippets.

intent-compliant contracts of this example.

Step 2: Simulate the configuration to identify all violations of the intent-compliant contracts. We start an iterative simulation of the original configuration. At each step, for each router, we examine the values of its contracts. If they are different from the values of the intent-compliant contracts, we record this violation, force the router's behavior to obey the contracts, and continue the simulation. As such, the simulation switches to a symbolic variant of the original configuration. In the end, the simulation yields the intent-compliant data plane in Step 1 as the output because (1) the whole simulation follows the intent-compliant contracts as invariants, and (2) the latter are the sufficient and necessary conditions for generating the intent-compliant data plane.

Figure 4 shows the simulation of our example. At one step, when C sends $[C,D]$ to B, the original configuration decides that C will set the corresponding $isExported$ to be $false$, which violates the corresponding intent-compliant contract. Therefore, we force the simulation to obey the contract and associate the corresponding route with the Boolean condition of this behavior (c1). As a result, B receives and selects $[B,C,D]$ as its path, then sends it to its neighbors. In the next step, when F receives route $[F,A,B,C,D]$ and compares it with $[F,E,D]$, F's configuration would set $isPreferred$ for this received route to be $true$. However, the intent-compliant contracts require F sets $isPreferred$ for the two paths comparison to be $true$, contradicting the original configuration. We again force the simulation to follow the intent-compliant contracts. As a result, F keeps $[F,E,D]$ as its path, instead of $[F,A,B,C,D]$, and also associate $[F,E,D]$ with the corresponding condition (c2). In the end, each router selects the path underlined in the Figure.

Step 3: Map violated contracts to configuration errors. For each violated contract, S^2Sim systematically maps it to its corresponding configuration snippet based on the type of violation and details such as routes and devices involved, as shown in Table 1. In our example, two contracts are violated. In the first case, C sets $isExported$ of $[C,D]$ to $false$. We map this violated contract to the export policy snippets on C for neighbor B, specifically the $filter$ policy (lines 3-5), which matches $[C,D]$ with the prefix filter and incorrectly discards it. In the second case, F sets $isPreferred$ of $[F,A,B,C,D]$ over $[F,E,D]$ to be $true$. We map this violated contract to the import policies on F that match $[F,A,B,C,D]$ and $[F,E,D]$, the $setLP$ policy of F (lines 3-9), where the first rule matches the AS-path of $[F,A,B,C,D]$ and sets the local preference to 200, while the second rule matches $[F,E,D]$ and sets its local preference to 80. Both snippets are consistent with the ground truth presented at the beginning of our example (§2).

Step 4: Generate conflict-free repair patches with contract-specific template-based constraint programming. Given a set of localized configuration errors, a naive repair approach is to recompute an intent-compliant configuration from the erroneous one using constraint programming (*e.g.*, network synthesis [9]). Specifically, we model the configuration as constraint C, where the parameters of erroneous snippets are symbolic, the violated contracts to fix as constraint V, and the non-violated contracts to preserve as constraint P. The repair patch computation is then expressed as finding a satisfiable assignment for $C \wedge V \wedge P$.

However, this approach has two main limitations: (1) encoding all configurations and contracts together leads to high computational cost, and, (2) contract conflicts on the same configuration snippet may cause unsatisfiable solutions. For example, if two contracts for different prefixes map to the same policy rule, with one requiring local preference >100 and the other <100, no single assignment can satisfy both, making the repair infeasible without changing the rule structure. To address these limitations, we introduce the *contract-specific template*, which uses fine-grained policy matching by adding a new rule that matches specific attributes of the route in the contract, such as prefix, community, and AS-path. This ensures that the rule only matches the intended route and does not affect others. The corresponding action is modeled as a symbolic variable, which is solved through constraint programming. This approach allows the repair problem to be decomposed into independent subproblems, and conflicts between contracts on the same configuration snippet are resolved, ensuring a satisfiable solution.

In this example, the contract-specific template for

$$\begin{array}{rcl} ints & ::= & int^* \\ int & ::= & (identifier,\ path\_req) \\ identifier & ::= & (srcDev,\ srcIp,\ dstDev,\ dstIp) \\ path\_req & ::= & (path\_regex,\ type,\ failures = K) \\ type & ::= & \textbf{any} \mid \textbf{equal} \end{array}$$

Figure 5: The syntax of S^2Sim's intent language.

$isExported$ of route $[C,D]$ inserts a new match rule before the currently matching one that uniquely matches this route. The action is modeled as a symbolic variable, which is solved through constraint programming to set the permit action as $True$. Likewise, for $isPreferred$ of routes $[F,A,B,C,D]$ and $[F,E,D]$, the template inserts a rule that matches $[F,A,B,C,D]$ and sets the permit action as $True$ and the local-preference to a value less than that of $[F,E,D]$ (< 80).

4 Basic Design

We now elaborate the basic design of S^2Sim with more details. For ease of presentation, we first focus on single destination prefix and describe how S^2Sim computes an intent-compliant data plane and derive the corresponding contracts (§4.1), and how it identifies all contract violations in the configurations using symbolic simulation and repairs them (§4.2). We then introduce how S^2Sim supports complex configurations such as route aggregation, ACL, multi-path routing (§4.3).

Intents. Figure 5 illustrates the syntax of the language S^2Sim uses to specify intents. Specifically, each intent is represented as a $(identifier,\ path\_req)$ tuple, where $identifier$ specifies the source and destination routers, and the $path\_req$ encodes the path requirement. A path requirement consists of a regular expression over devices ($path\_regex$), a type specifier (**any** or **equal**), and a failure scenario specifier that requires the intent to hold under up to arbitrary K link failures.

This syntax is expressive enough to capture common intents such as reachability, waypoint, and multi-path reachability. For example, in Figure 1, the reachability intent from router B to router D is specified as $(B.^*D, \textbf{any}, failures = 0)$, and the waypoint intent through router C is specified as $(A.^*C.^*D, \textbf{any}, failures = 0)$. A multi-path reachability intent can be specified as $(S.^*D, \textbf{equal})$, indicating that all available paths from S to D should be used equally.

4.1 Intent-Compliant Contracts

Intent-compliant contracts are predicates on router behaviors such that, if satisfied, the network will yield an intent-compliant data plane. Given a set of intents and an erroneous configuration, we find such contracts in two steps: (1) compute an intent-compliant data plane with small differences from the data plane generated by the erroneous configuration; and (2) derive intent-compliant contracts accordingly.

Compute an intent-compliant data plane by finding paths for unsatisfied intents. Suppose the data plane generated by the erroneous configuration satisfies m intents and violates n intents. We use the valid paths in the erroneous data plane for those m intents as an initial set of path constraints. For each of the n unsatisfied intents, we find it a shortest valid path overlapping with existing constraints as mush as possible (*i.e.*, preferably having a superpath/subpath relation) and without breaking the existing constraints (*i.e.*, the founded path must not form any loop with paths in the path constraints), and add this path into the set of path constraints. We find such a valid path using deterministic finite automaton (DFA) multiplication. During the iteration, given an intent x, if no path satisfying x and the path constraints can be found, we gradually remove paths from the path constraints one at a time, mark the corresponding intents as unsatisfied, and try to find a valid path for x satisfying the new set of path constraints. This backtracking continues until such a path is found and then we move on to the next unsatisfied intent.

To make sure we can quickly find an intent-compliant data plane, we cope with the combinatorial complexity of the order of path finding for unsatisfied intents and the order of path constraints backtracking with the following two principles:

Path finding for unsatisfied intents: more constrained intents first and recently backtracked intents first. To be concrete, more constrained intents refer to ones that require not only reachability, but sequence of routers in paths (*e.g.*, waypointing). We try to find shortest valid paths for them with priority because their sets of intent-compliant paths are typically smaller than those of intents on only reachability, and more likely become empty when there are too many path constraints to satisfy simultaneously. For example, suppose none of the intents in the example network (Figure 1) are satisfied, we prioritize computing paths for A and F's intents over all other routers' intents. Next, recently backtracked intents refer to ones whose paths are recently removed from the path constraints to increase the chance for finding a valid path for an unsatisfied intent. We find paths for them with priority because their sets of intent-compliant paths just shrank due to the recent removal of their valid paths.

Path constraints backtracking: closest path first and newest added path first. Supposed for intent x, we could not find an intent-compliant path without breaking the existing path constraints. We examine the paths in the path constraints and remove the one whose source router is closest, in terms of hop count, to the source router of x. For example, when backtracking to compute A's intent, we first remove the closer paths of B and F instead of the farther ones like C or E. Our rationale is that by removing such a closest path, we have a better chance for finding x a valid path. Next, the concept of newest added path is self explanatory. By removing the newest added path, we seek a valid path for x without causing too many changes to the set of existing valid paths.

Although these principles are not perfect in theory and may require computing intent-compliant paths in factorial orders of intents in the worst case. In practice, the process efficiently computes an intent-compliant data plane for real and synthetic networks of various scales (§7).

Derive intent-compliant contracts via path existence conditions. An intent-compliant data plane can be decomposed into a set of intent-compliant contracts of routers via the sufficient and necessary conditions of path existence. Specifically, given a path $R_1 \to \ldots, \to R_{i-1} \to R_i \to R_{i+1}, \ldots, \to R_n]$, it exists in the data plane if and only if for each router R_i, it peers with R_{i+1}, accepts and prefers $R_{i+1}, \to, \ldots, \to, R_n$ over all other routes, and announces $R_i, \to, R_{i+1}, \ldots, \to, R_n$ to R_{i-1}. Thus, to ensure a path in the intent-compliant data plane exists, each router on the path must satisfy contracts related to peering, route selection, and announcement behaviors.

4.2 Diagnose and Repair Errors

With the intent-compliant contracts derived from the data plane, S^2*Sim* repairs configuration errors in three steps: (1) identifying all violated contracts using selective symbolic simulation; (2) localizing errors by mapping violated contracts to configuration snippets; (3) computing conflict-free repair patches using template-based constraint programming.

Identify violated contracts using selective symbolic simulation. Given the set of intent-compliant contracts and the original configuration, S^2*Sim* simulates the protocol behavior. At each step, it checks whether the router's behavior—such as importing, exporting, preferring, or peering—matches the corresponding contract. If a behavior violates the expected contract, S^2*Sim* forces the behavior to conform to the contract and selectively switches to the symbolic variant of the configuration where the violation has been corrected. The violated contract is then attached to the corresponding route as an annotation. This selective symbolic simulation proceeds iteratively until the resulting data plane converges to the intent-compliant one. In the end, S^2*Sim* collects all encountered violated contracts for error localization and repair.

Although most configurations can be handled in a prefix-independent manner for path vector protocols, which enables independent symbolic simulation for each prefix, the establishment of a peering session for one prefix can affect the propagation of all prefixes, as sessions are established at the router level. Therefore, S^2*Sim* treats the *isPeered* contract as shared and enforces it across all prefixes during symbolic simulation: if any prefix requires a peering session on a given link, S^2*Sim* forces all prefixes to establish that peer relationship.

Localize configuration errors by mapping violated contracts to configuration snippets. Given all violated contracts identified through selective symbolic simulation, along with details such as routes and devices, S^2*Sim* maps each violation to its corresponding erroneous configuration snippet (as shown in Table 1). For example, a violated *isImported* contract on device u, involving neighbor v and route r with attributes such as AS-path, community, and next-hop, is mapped to the import policy rule on u that matches r from v. Similarly, a violated *isPeered* contract is mapped to the relevant neighbor statements and associated interface configurations on the involved devices. This fine-grained mapping pinpoints the exact configuration locations needing repair.

Repair configuration errors using contract-specific template-based constraint programming. Given all localized configuration errors, the direct approach to repair them is modeling the repair process as a constraint-solving problem. Specifically, the parameters in the erroneous configuration snippets are symbolized, forming a partially specified configuration, denoted as C. Let V represent the violated contracts to be fixed and P represent the non-violated contracts to be preserved. The repair problem is encoded as a constraint problem of finding a satisfiable assignment for $C \land V \land P$, aiming to find parameter assignments that satisfy all contracts.

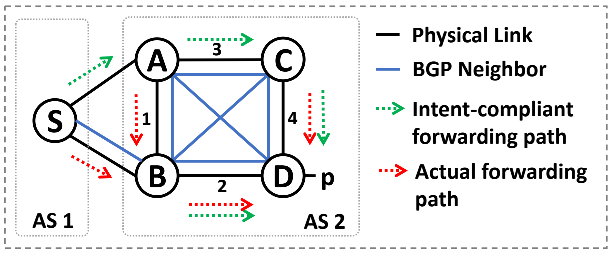
However, naively encoding all contracts and configurations together is computationally expensive, and may lead to unsatisfiable assignments due to conflicts between contracts for different prefixes mapped to the same configuration snippet. For policy-related contracts (e.g., *isImported*, *isExported*, *isPreferred*), S^2*Sim* utilizes contract-specific templates which leverage BGP's fine-grained policy control to exactly match the route. We summarize the templates used for each contract in Appendix B. This involves adding a rule that matches the specific route before the current matched one, ensuring the patch only affects the desired route. Such a design allows independent repair of each contract, resolving conflicts between contracts for different prefixes on the same configuration snippet and ensuring a solution. Constraint programming is then used to find a satisfiable value for the action parameter to correct the contract. For the *isPeered* contract, S^2*Sim* resolves dependencies during symbolic simulation. Patches can be computed independently using a template that models the peer configuration, with constraint programming finding the necessary assignments to establish the peer relationship.

4.3 Support Complex Configuration

To support complex configurations, S^2*Sim* extends mechanisms for route aggregation, ACLs, and ECMP.

Route aggregation. Route aggregation combines multiple IP prefixes into a single, larger prefix to reduce routing table size and improve efficiency. The main challenge is ensuring the aggregated prefix's propagation path satisfies all contracts from its sub-prefixes. To address this, S^2*Sim* solves the contracts collectively, rather than independently for each sub-prefix. By considering all contracts together and using template-based repair for the involved prefixes, S^2*Sim* creates a repair patch that consistently satisfies all contracts, avoiding discrepancies in the aggregated route's propagation path. If no solution is found, S^2*Sim* employs a disaggregation strategy, splitting the aggregated prefix into its original components so that each prefix's contract can be satisfied individually.

Access control list. The ACL allows a router to block traffic in the data plane. To support this, S^2*Sim* introduces two additional contracts, *isForwardedIn* and *isForwardedOut* (as



(**a**) Intent-compliant data plane.

Intent-compliant Contracts for OSPF	
isEnabled(C, D) == true	isPreferred (C, [C, D], *) == true
isEnabled(B, D) == true	isPreferred (B, [B, D], *) == true
isEnabled(A, C) == true	isPreferred (A, [A, C, D], *) == true

(**b**) Intent-compliant contracts for the OSPF network.

Figure 6: The multi-protocol example network.

shown in Table 1), which specify whether packets destined for a given prefix are allowed to enter or leave the router along the intended forwarding paths. During symbolic simulation, S^2Sim checks these contracts by comparing ACL behavior with the intent-compliant data plane. If a packet that should be forwarded is blocked by an ACL, the corresponding contract is marked as violated. Due to the fine-grained nature of ACL rules, such violations can be diagnosed and independently repaired using contract-specific templates within the constraint-solving framework.

Multi-path routing. Multi-path routing is a strategy where packet forwarding to a single destination occurs over multiple equal-priority paths (ECMP). To support this, S^2Sim introduces two extensions. First, when computing the intent-compliant data plane, it records all equally preferred paths, allowing a node to have multiple next hops and ensuring ECMP semantics are preserved. Second, S^2Sim introduces a new contract, *isEqPreferred*, specifying two routes be equally preferred at a given node. When this contract is violated, S^2Sim identifies the configuration error and repairs it independently, similar to how *isPreferred* contracts are handled.

5 Multi-Protocol Networks

After describing S^2Sim's basic design using BGP as an example, we introduce how it diagnoses and repairs networks where multiple protocols (*e.g.*, OSPF and BGP) co-exist. Because S^2Sim can handle a network running different routing protocols in different regions by examining these regions separately, we focus on networks running different routing protocols in a layered setting (*e.g.*, OSPF as underlay and BGP as overlay). Error diagnosis and repair for such networks is more challenging due to the dependency between these protocols.

To this end, we present our solution for multi-protocol networks in §5.1, and describe how to extend the selective symbolic simulation approach to diagnose and repair link-state routing protocols in §5.2.

5.1 Our Solution

Inspired by modular verification approaches such as Kirigami [34] and Lightyear [32], we use an assume-guarantee approach, and decompose the intents of the multi-protocol network into distinct layers. The approach first assumes the proper functioning of the underlay network (e.g., reachability between overlay-neighboring routers), and diagnosis and repairs the overlay network to be intent-compliant. It then uses the assumption as the intents for the underlay network and repeat the diagnosis and repair process. We use the following example to illustrate our key ideas.

Example. Figure 6a shows an example of two ASes peered with eBGP and routers A, B, C, and D are connected using OSPF in underlay and peered with iBGP in a full mesh in overlay. In the configuration, S is only peered with B via eBGP and the OSPF cost of each link in AS 2 is shown in the edges. The destination IP prefix p is located at node D and is advertised via BGP. The network intents are: (1) all routers can reach prefix p, and (2) the path of S to reach p should not pass B. The data plane of the original configuration is marked in red, which is erroneous. The actual forwarding path [S, B, D] violates Intent 2. This is due to two configuration errors: first, S lacks a BGP peer with A; second, misconfigured OSPF costs in AS 2 cause A to prefer reaching D via B instead of C.

Derive intents for overlay and underlay. We begin by verifying the specified intents against the generated data plane, classifying them as either satisfied or violated. These intents are then decomposed into overlay and underlay intents.

In this example, Intent (1) is satisfied and corresponds to the overlay. Intent (2) is violated and must be split into overlay and underlay components. To satisfy it, we compute the shortest valid path in the physical topology and determine that S must reach D via $[S,A,C,D]$. Based on the domain knowledge that iBGP nodes do not re-advertise routes learned from other iBGP peers (i.e., path $[A,C,D]$ cannot be learned via iBGP), and noting that A, C, and D belong to the same IGP domain, we derive the following sub-intents: 1) BGP Intent 1: S reaches D via $[S,A,\cdots,D]$, and 2) OSPF Intent 1: A reaches D via the path $[A,C,D]$. Additionally, since BGP relies on OSPF to establish neighbor sessions using OSPF-learned IPs, even between non-directly connected nodes (e.g., A and D), we derive OSPF Intent 2: D and its neighboring routers A, B, and C must be mutually reachable.

Process each layer independently. Given the derived BGP intents, we generate the BGP intent-compliant data plane, and selectively simulate the BGP network to find violated contracts. In this example, we identify a violated *isPeered* contract between S and A, and generate a repair that establishes the missing peer. Similarly, using the derived OSPF intents, we build the OSPF intent-compliant data plane, and then apply selective symbolic simulation to diagnose and repair the underlay network, as provided in §5.2.

5.2 Link-State Routing Protocols

The fundamental difference between path-vector and link-state protocols is that the former selects the best paths for different prefixes independently (*i.e.*, per-prefix policy routing) while the latter does so with a unified metric (*i.e.*, prefix-oblivious link costs). Thus, we must first collect all OSPF intents across different prefixes before deriving the intent-compliant contracts. Then, we follow the design of path-vector protocols in §4 to handle link-state routing protocols.

Define contracts. Unlike BGP, OSPF does not establish peers but instead relies on OSPF-enabled interfaces. Additionally, OSPF does not support route policies and determines routing paths solely based on link costs. Accordingly, we define two types of contracts for OSPF: 1) $isEnabled(a,b)$: the interfaces between nodes a and b are OSPF-enabled, and 2) $isPreferred(u,r,r')$: identical to the BGP definition, it specifies that router u should prefer route r over r'.

Derive intent-compliant data plane, and perform selective symbolic simulation to find violated contracts. Given the set of underlay intents, we compute the corresponding intent-compliant data plane and derive the associated contracts. Figure 6b illustrates the contracts for OSPF in Figure 6a, where $isEnabled$ contracts are derived from OSPF Intent 2, and $isPreferred$ contracts are inferred from the required forwarding behavior at nodes A, B and C.

To reuse the simulation method to diagnose configuration errors, we simulate OSPF using a path-vector abstraction, where path selection is based on the cumulative cost. In this example, the diagnosis reveals a contract violation at node A, which incorrectly prefers the path $[A,B,D]$ over $[A,C,D]$, indicating that the OSPF cost configuration is incorrect.

Model the repair process as a MaxSMT problem. In OSPF networks, contract violations include enabled and preference errors. OSPF-enabled error (*i.e.*, violations of $isEnabled$ contracts) can be resolved by enabling OSPF on the relevant interfaces. Preference error (*i.e.*, violations of $isPreferred$ contracts), however, requires adjusting link costs. Since OSPF computes a single forwarding tree per destination, modifying link costs for one path may inadvertently affect the forwarding behavior of other nodes.

To address the issue, we adopt the constraint programming approach to carefully adjust link costs while preserving as much of the original configuration (costs) as possible. Specifically, we encode the OSPF network and its associated contracts into a MaxSMT formulation. Hard constraints ensure that violated contracts are repaired and that non-violated contracts remain satisfied. Soft constraints aim to preserve the original link costs to minimize configuration changes. In this example, we encode the repair problem as:

$$
\begin{aligned}
(hard)\quad & \{l_{CA}+l_{AB}+l_{BD}>l_{CD}\}\wedge\{l_{BA}+l_{AC}+l_{CD}>l_{BD}\}\\
& \wedge\{l_{AB}+l_{BD}>l_{AC}+l_{CD}\}\\
(soft)\quad & l_{AB}==1\wedge l_{BD}==2\wedge l_{AC}==3\wedge l_{CD}==4
\end{aligned}
$$

where $l_{a,b}$ denotes the link cost between nodes a and b. In the proposed example, a valid repair solution is to update l_{AB} to 7, while keeping all other link costs unchanged.

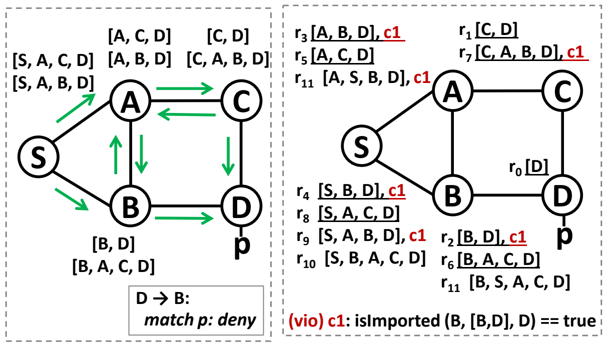
6 Tolerating K-link Failure

This section presents our designs to provide fault-tolerant capability. We focus on the k-link failure reachability, which refers to the network's ability to preserve connectivity between a given source and destination node pair in the presence of up to k arbitrary link failures.

6.1 Key Idea

Our key idea is to construct a fault-tolerant data plane where at least one path exists for the forwarding traffic under arbitrary k link failure(s). Specifically, we compute $(k+1)$ edge-disjoint paths for each k-link failure tolerance intent. Two edge-disjoint paths mean that the edges of them do not intersect. Since these paths are edge-disjoint, each link failure results in, at most, one path failing to forward the packet. According to the Pigeonhole Principle, at least one path must exist to forward the packet when k links fail.

For multi-protocol networks, we use a similar approach in §5 to decouple the derived fault-tolerant data plane into overlay intent-compliant data plane and underlay intent-compliant data plane. Then we derive intent-compliant contracts and perform selective symbolic simulation to diagnose and repair errors for each layer separately.



(a) The intent-compliant fault-tolerant data plane. **(b)** The symbolic simulation with single-link failures.

Figure 7: The single-link failure tolerance example network.

6.2 Design Details

As discussed in Section 5.2, a link-state routing protocol can be simulated as a path-vector protocol, with the key difference being that link-state protocols compare only path costs and do not support routing policies. Therefore, we focus on BGP to illustrate the design details.

Example. Figure 7 illustrates an example network to ensure single-link failure tolerance, where the intent is that all routers must be able to reach prefix p under any single-link failure. In this network, five routers are connected via eBGP, all using the default BGP configuration except for router B, which is configured to drop routes from neighbor D that match prefix p. The intent is satisfied under normal conditions and remains valid under certain single-link failures.However, for failures such as (C,D) or (A,C), since B drops D's route, this breaks the reachability intent under these failure scenarios.

Derive fault-tolerant data plane and contracts. For each k-link failure tolerance intent, we compute $k+1$ edge-disjoint paths and merge them into a fault-tolerant data plane. These $k+1$ paths are obtained iteratively by invoking the shortest path algorithm $k+1$ times, each time removing the edges computed in the previous iteration from the graph. Figure 7a shows the merged fault-tolerant data plane of the example network with single-link failure tolerant, where each node is associated with two edge-disjoint paths, indicating that the traffic must be forwarded in either of the paths. For example, $[B,A,C,D]$ and $[B,D]$ are the derived data plane for B to ensure reachability to p even under single-link failure.

In the intent-compliant data plane where each node may have multiple forwarding paths, we derive the *isPeered*, *isExported* and *isImported* contracts to ensure the existence of these paths, along with *isPreferred* contracts to guarantee their selection over non-forwarding alternatives. However, we do not derive *isPreferred* contracts that impose an order among the forwarding paths themselves, as the specific choice among them is irrelevant for satisfying the reachability intent. In this example, the most important intent-compliant contracts for node B are that both $[B,D]$ and $[B,A,C,D]$ are preferred over all other available routes.

Selective symbolic execution to diagnose and repair errors. According to the fault-tolerant contracts, we simulate the router to select and propagate multiple routes to diagnose and repair errors as in previous designs. In this example, as Figure 7b shows, when B imports the route $[B,D]$ from D, we find that the *isImported* contract for this route is violated, so we record this violation as an error and continue the simulation. After that, when B imports the route $[B,A,C,D]$, we find it is intent-compliant and select both $[B,D]$ and $[B,A,C,D]$ as B's best routes and send them to neighbors. In the end, C, A, and S also select and send multiple routes that comply with their intents. Thus, we can simulate the network and diagnose the error under different failure scenarios.

6.3 Discussion

Beyond reachability intents, we also handle other non-trivial intents (*e.g.*, waypoint and avoidance) under arbitrary k-link failures, following the same approach used for reachability. For each intent, we compute multiple compliant paths and ensure that at least one remains available across all failure scenarios.

However, constructing a fault-tolerant data plane for multiple non-trivial intents may lead to path explosion. Each such intent requires computing $k+1$ edge-disjoint paths while simultaneously preserving the path constraints imposed by other intents. In some cases, this necessitates backtracking to exhaustively enumerate valid path combinations across intents to find a satisfiable solution. Developing more efficient algorithms to compute fault-tolerant data planes is our future work. Different from waypoint or avoidance intents, supporting multiple k-link failure reachability intents does not cause path explosion. This is because such intents are always handled last (as the principle one in §4.1), and their compliant paths do not break existing path constraints, thereby avoiding the need for backtracking.

In practice, users may require path preference under failures, where some paths are designated as primary and others as backups. The current design can be extended to support such path preference intents, given an ordered set of paths (*e.g.*, $P_1 \gg \cdots \gg P_x$). S^2*Sim* can first ensure these paths are available and then enforce the preferences by adding *isPreferred* contracts to the intersection nodes along these paths.

Configurations Features		Real		Synthesized		
		DC-WAN	IPRAN	WAN	IPRAN	DCN
Routing Protocol	BGP	+	+	+	+	+
	ISIS	-	+	-	-	-
	OSPF	+	-	-	+	-
	Static Route	+	+	+	+	+
Routing Policy (Filter)	Prefix-list	+	+	+	+	-
	As-Path-list	+	-	-	-	-
	Community-list	+	+	-	+	-
Routing Policy (Modifier)	Set Local-preference	+	+	-	+	-
	Set Community	+	+	-	+	-
Routing Control	Route Aggregation	+	-	-	-	-
Traffic Control	Access Control List	+	-	+	-	-
	Equal-Cost Multi-Path	-	-	-	-	+

Table 2: Configuration features of the evaluated networks. A + (–) indicates the feature is present (absent) in the network.

7 Performance Evaluation

We have implemented a prototype of S^2*Sim* in 8K LoC in Java as a plugin of Batfish[13], and will open source it upon publication. All experiments were conducted on a Linux server equipped with two Intel Xeon Silver 4210R 2.40GHz CPUs and 128GB of DDR4 DRAM.

Our evaluation answers the following two questions: (1) Can S^2*Sim* and SOTA tools diagnose and repair real network configuration errors? (§7.1) (2) How efficiently S^2*Sim* diagnose and repair the errors for large-scale networks? (§7.2)

Configurations. We use both real network configurations and synthesized network configurations for evaluation. Table 2 summarizes the configuration features of these networks.

(1) Real network configurations. One major public cloud provider offers an operating wide-area network (WAN) consisting of 88 nodes interconnecting data centers (i.e., *DC-WAN*), with an average of 5K lines of configuration per node. Another major vendor provides four IP radio access network (i.e., *IPRAN1-IPRAN4*) with 36, 56, 76, 106 nodes, connecting base stations to base station controllers via routers, with an average of 300 lines of configuration per node. Both DC-

Categories	Configuration Error Types		IPRAN	DC-WAN	DCN	CPR	CEL	S²Sim
Redistribution (1)	Missing redistribution command for the static or connected route	(1-1)	+	+	+	√	√	√
	Extra prefix-list filters the route during redistribution	(1-2)	+	+	+	×	√	√
Propagation (2)	Incorrect prefix-list filters the route during propagation	(2-1)	+	+	-	√	√	√
	Incorrect as-path/community-list filters the route during propagation	(2-2)	-	+	-	×	×	√
	Omitting permitting a route with specific prefix	(2-3)	+	+	-	√	√	√
Neighboring (3)	OSPF is not enabled on the interface	(3-1)	+	+	-	√	√	√
	Missing the BGP neighbor statement	(3-2)	+	+	+	√	√	√
	Missing *ebgp-multihop* for indirectly-connected eBGP neighbors	(3-3)	+	-	-	×	×	√
Preference (4)	Incorrectly setting a higher local-preference for the non-preferred path	(4-1)	-	+	-	×	×	√
	Omitting setting a higher local-preference for the preferred path	(4-2)	-	+	-	×	×	√

Table 3: Errors introduced in synthesized configurations, all of which are observed in real networks. A + (–) indicates the presence (absence) of the error in the real network. A ✓ (×) indicates whether the tool is able (unable) to handle the error.

WAN and IPRAN utilize the underlay-overlay architecture. Additionally, the DC-WAN configuration provider also offers the configuration features of their operating DCNs. However, due to the large size of real DCN configurations, the corresponding configuration files are not available to us.

(2) Synthesized network configurations. We adopt the real WAN topologies from TopologyZoo (33 to 155 nodes), and use NetComplete [9] to generate WAN configurations with configuration lines 3K to 13K. We also consider two large-scale networks: IPRANs having nodes ranging from 1000 to 3000 and configuration lines from 176K to 561K, and data-center networks (DCN) using the fat-tree architecture, having nodes ranging from 80 to 1280 and configuration lines from 2K to 140K. We synthesize IPRAN/DCN configurations following the structure of real IPRAN/DCN networks.

Intents. We consider three types of intents for evaluation: (1) Reachability (RCH): a source node can reach a destination node without link failures, (2) Fault-tolerant reachability: a source node can reach a destination node with k-link failures, and (3) Waypoint reachability (WPT): a source node can reach a destination node via specific waypoint node(s).

7.1 Comparison with Existing Tools

We compare S^2*Sim* with CPR [14] and CEL [15] in both capability and running time. As discussed in Section 2, other tools such as Batfish, Minesweeper, ACR are not adequate for configuration diagnosis or repair. , which is why we select CPR (for repair) and CEL (for diagnosis) as our baselines. However, since CPR and CEL lack support for certain features present in real configurations, we construct synthesized WAN configurations and inject errors supported by these tools but derived from real-world cases, to compare their running time.

Errors in Real Configurations. Table 3 summarizes common configuration errors observed in real DC-WAN and IPRAN networks. Additionally, the DC-WAN configuration provider has reported a list of typical errors found in their managed DCNs. We categorize these errors into four types. Redistribution-related and neighborhood-related errors often stem from missing configurations for route redistribution or the establishment of routing adjacency. These are the most frequently encountered issues across all real network configurations. Route propagation-related and preference-related errors typically result from incorrect or absent routing policies (e.g., filters), and are found only in DC-WAN and IPRAN networks. This is because these two network types rely heavily on policy-based routing for inter-domain traffic control, whereas DCNs generally do not require such mechanisms.

Comparison. We evaluate CEL and CPR on real-world network configurations and find that CPR lacks support for AS-path/community filters, the local preference modifier, and multi-protocol networks with the underlay–overlay architecture, while CEL similarly fails to handle regular-expression-based AS-path, community filters and the local-preference modifier present in these configurations.

To verify accuracy, we use the example network in Figure 1 and inject each real-world error from Table 3 one at a time to evaluate the capabilities of existing tools. The results show that CPR can only diagnose and repair 5 out of the 10 injected errors, while CEL can diagnose 6. Notably, the errors these tools fail to handle are all commonly found in real DC-WAN and IPRAN configurations. In contrast, S^2*Sim* successfully supports all listed error types. Figure 8 shows the runtime of S^2*Sim* on real network configurations for various intent checks, including reachability, waypoint, and fault tolerance. The results demonstrate S^2*Sim*'s effectiveness and efficiency in handling feature-rich scenario. In the figure, the first simulation time refers to the runtime of the protocol simulator to generate data plane, common to all simulator-based tools, while the second simulation time corresponds to the selective symbolic simulation performed specifically by S^2*Sim*.

To further evaluate performance, we synthesize WAN configurations based on five real-world topologies from TopologyZoo, ranging from 33 to 155 nodes. In each network, we randomly inject between one and five errors from the five error types supported by CEL and CPR. We define three sets of intents: S1 (2 RCH + 2 WPT), S2 (6 RCH + 2 WPT), and S3

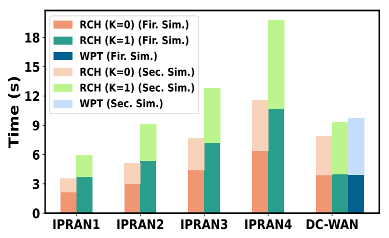
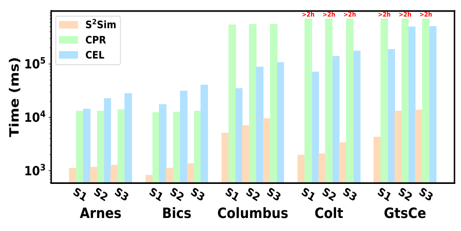
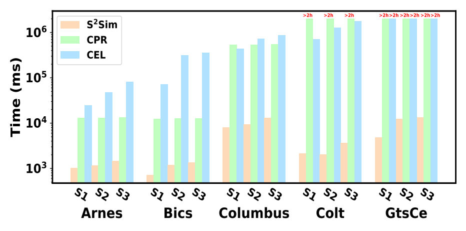


Figure 8: The runtime of S^2Sim on five real configurations, including two rounds of simulation.

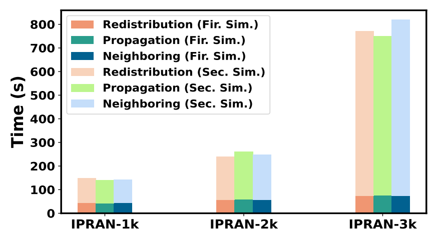


(a) Reachability.

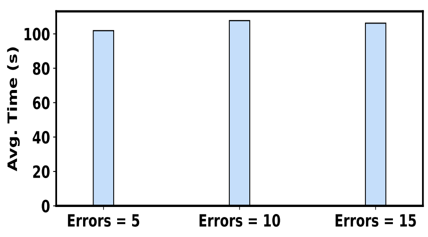


(b) Fault-tolerant reachability (k=1).

Figure 9: The runtime of S^2Sim and other tools on synthesized WAN configurations.



(a) Error category vs. runtime.



(b) Error number vs. runtime.

Figure 10: Impact of error types and error number on S^2Sim runtime in IPRANs.

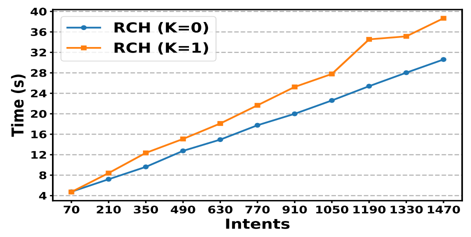


Figure 11: Impact of intent number on S^2Sim runtime in a DCN.

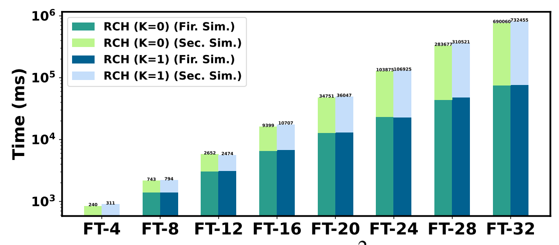


Figure 12: Impact of net. scales on S^2Sim runtime in DCNs.

(10 RCH + 2 WPT). Each injected error is crafted to violate at least one intent. Details on the network scales, injected errors, and intents are summarized in Table 4 in Appendix C.

These synthesized configurations preserve realistic complexity, incorporating variations in network size, number of errors, and number of intents. Figure 9 compares the runtime of all tools across different network sizes and fault-tolerant scenarios. The results show that S^2Sim achieves over > 10X speedup compared with CEL and CPR in diagnosing and repairing configuration errors under both no-link failure and single-link failure conditions. For large networks with 150+ nodes, CPR fails to complete the repair even for non-link-failure reachability, and CEL fails to complete the diagnosis of 1-link failure tolerance reachability.

7.2 Scalability of S^2Sim

We use a collection of synthesized IPRAN (multi-protocol) and DCN (single-protocol) configurations with increasing size to test the scalability of our tool. We evaluate how the error type, error number, intent number, and network scale impact S^2Sim's performance.

The error type and error number have negligible impact on runtime. We choose IPRANs for evaluation due to their broader variety of injected errors compared with DCN. We first vary the error type, while keeping the error number (*i.e.*, 1) and the number of intents (*i.e.*, 1) constant, to evaluate S^2Sim's performance. We randomly inject one representative error from each IPRAN-related error category across 1K-, 2K-, and 3K-scale networks. As Figure 10a shows, the diagnosis and repair time remain nearly constant across all error categories for each network. This is due to the fact that we model each contract as a Boolean condition, which requires nearly constant time to check the violations of different contracts.

Since the error type has little impact on performance, we do not differentiate error type, and inject errors from all error types into the 1K-scale IPRAN with 10 intents. Figure 10b shows the average runtime when the error number varies. We can find that the runtime is nearly constant across different numbers of errors, indicating that the error number has negligible impact on runtime.

The number of intents has a linear impact on runtime. We use a small-scale synthesized DCN (*i.e.*, FT-8 with 80 nodes) for evaluation. Given that the error type and number are not fatal factors impacting performance, we randomly inject 10 errors into this network. We vary the number of intents that the network must satisfy. Figure 11 shows the runtime required to diagnose and repair the network for both reachability and fault-tolerance reachability. We see a linear increase in repair times as the number of intents increases. This stems from two facts: (1) each new (unsatisfied) intent requires computing a new intent-compliant path, and (2) each intent-compliant path derives new contracts. Thus, the time required to compute the intent-compliant data plane and the time spent checking contract violations increase as the number of intents increases. Furthermore, fault-tolerant reachability requires computing more paths and generating more contracts per intent, leading to a faster increase in runtime.

The network scale has quadratic impact on runtime. We use synthesized DCNs for evaluation. We vary the number of nodes while keeping the number of intents constant (*i.e.*, 10), and the results are shown in Figure 12. Although the overall runtime appears to grow exponentially with the network size, the majority of this increase comes from the first simulation, which is common to all simulation tools and should not be attributed to our system's overhead. In contrast, the second simulation, the core component of S^2*Sim*, exhibits quadratic growth. Results in Figure 10a (*i.e.*, IPRAN) show a similar trend. Additionally, we observe that the runtime of reachability and fault-tolerant reachability intents remains comparable. This is because computing one versus multiple intent-compliant paths introduces similar overhead in topologies that are highly symmetric and redundant, such as fat-trees.

8 Related Work

Network synthesis. Propane [6] uses DFA to generate an intent-compliant data plane based on the intent. S^2*Sim* adopts this idea but proposes several principles to compute an intent-compliant data plane with small differences from the one generated by the erroneous configuration. NetComplete [9], Merlin [30], and AED [1] symbolize parameters of the erroneous configuration and build a global constraint programming model to compute an intent-compliant configuration, which faces a search space explosion and potential unsatisfiable assignments due to contract conflicts. To address these issues, S^2*Sim* uses contract-specific templates to enable per-violated-contract constraint programming.

Network diagnosis and repair. Existing tools face various limitations. CEL [15] uses an SMT-based model and computes the minimal correction set for error localization, but it cannot handle AS-path due to space explosion issues. CPR [14] employs constraint programming to repair configuration errors but does not support local-preference, a commonly used attribute, due to graph abstraction. ACR [26] relies on coverage-based tools to identify erroneous configurations and repair errors using human experience, requiring multiple trials and potentially missing real errors. In contrast, S^2*Sim* proposes designs such as intent-compliant contract derivation, selective symbolic simulation, and contract-specific templates, enabling greater efficiency and accuracy.

Configuration error localization. Campion [33] localizes configuration errors by comparing the differences between the two given configurations, SelfStarter [21] and Diffy [22] detect potential configuration errors by identifying outliers among a set of configurations. These works focus on finding errors across multiple router configurations, whereas our work targets automatically diagnosing and repairing intent violations within a single network configuration.

Symbolic execution. A large body of prior work has applied symbolic execution to network verification or protocol verification. For example, HSA [23], APV [41], and SymNet [31] symbolize the packet space to verify whether the network satisfies specific properties such as reachability or loop-free. Building on this idea, Hoyan [43], SRE [45], and Expresso [35] symbolize the link-state space or external route space to verify whether the network satisfies specific properties under different environments. In contrast, our approach leverages symbolic execution to diagnose and repair configuration errors, shifting the focus from merely checking whether a property holds to identifying the root cause of violations and enabling corrective actions.

9 Limitations

Inability to guarantee intent-compliant convergence in nondeterministic scenarios. S^2*Sim* leverages the intent-compliant data plane to repair configuration errors. However, it cannot handle nondeterministic convergence scenarios, where a repaired configuration may have multiple converging states, such as BGP wedgies [18]. In such cases, we cannot guarantee that the repaired configuration will always converge to the intent-compliant data plane.

Non-minimal repair. Our repair method does not guarantee minimal changes to the configuration. It simply minimizes differences in the data plane, assuming this leads to fewer violated contracts and smaller configuration updates. However, small data-plane changes may still trigger large configuration edits. Achieving minimal repair remains an open problem.

Scalability for link-state protocols. Our current solution for link-state protocols relies on global constraint solving and cannot leverage local templates, as used in path-vector protocols, to reduce the complexity and optimize performance.

10 Conclusion

This paper presents S^2*Sim*, a novel system for automatic routing configuration diagnosis and repair via the following new designs: 1) deriving intent-compliant data plane and contracts from erroneous configurations, 2) finding violated contracts and diagnosing errors via symbolic simulation, and 3) repairing errors with constraint programming. Extensive experiments demonstrate its efficiency and efficacy.

This work does not raise any ethical issues.

Acknowledgments

We sincerely thank our shepherd, Ryan Beckett, and the anonymous NSDI reviewers for their valuable feedback on this paper. We also thank Ruzica Piskac and Thomas Wies for their help during the preparation of this paper. This work is supported in part by the National Key R&D Program of China 2022YFB2901502, the National Natural Science Foundation of China (Grant No. 62532010, 62572408, 62172345, and 62302409), and the Fundamental Research Funds for the Central Universities (Grant No. 20720250058 and 20720240049).

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *CoNEXT '20*, pages 482–495. ACM, 2020.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *NSDI '20*, pages 201–219. USENIX, 2020.

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *SIGCOMM '17*, pages 155–168. ACM, 2017.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *SIGCOMM '18*, pages 476–489. ACM, 2018.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *POPL '19*, pages 1–27. ACM, 2019.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM '16*, pages 328–341. ACM, 2016.

[7] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *SIGCOMM '23*, page 122–135. ACM, 2023.

[8] Ang Chen, Chen Chen, Lay Kuan Loh, Yang Wu, Andreas Haeberlen, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Data center diagnostics with network provenance. *IEEE Data Engineering Bulletin*, 41(1):74–85, 2018.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *NSDI '18*, pages 579–594. USENIX, 2018.

[10] Xing Fang, Feiyan Ding, Bang Huang, Ziyi Wang, Gao Han, Rulan Yang, Lizhao You, Qiao Xiang, Linghe Kong, Yutong Liu, et al. Network can help check itself: Accelerating smt-based network configuration verification using network domain knowledge. In *INFOCOM '24*, pages 2119–2128. IEEE, 2024.

[11] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI '16*, pages 217–232. USENIX, 2016.

[12] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *NSDI '05*, pages 43–56. USENIX, 2005.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI '15*, pages 469–483. USENIX, 2015.

[14] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *SOSP '17*, pages 359–373. ACM, 2017.

[15] Aaron Gember-Jacobson, Ruchit Shrestha, and Xiaolin Sun. Localizing router configuration errors using minimal correction sets. *arXiv:2204.10785. Retrieved from https://arxiv.org/abs/2204.10785*, 2022.

[16] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM '16*, pages 300–313. ACM, 2016.

[17] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. Efficient verification of network fault tolerance via counterexample-guided refinement. In *CAV '19*, pages 305–323. Springer, 2019.

[18] Tim A. Griffin and Geoff Huston. BGP Wedgies. RFC 4264, November 2005.

[19] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *NSDI '20*, pages 701–721. USENIX, 2020.

[20] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. Validating datacenters at scale. In *SIGCOMM '19*, pages 200–213. ACM, 2019.

[21] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *NSDI '20*, pages 999–1013. USENIX, 2020.

[22] Siva Kesava Reddy Kakarla, Francis Y Yan, and Ryan Beckett. Diffy: Data-driven bug finding for configurations. pages 199–222, 2024.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI '12*, pages 113–126. USENIX, 2012.

[24] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xian-

long Zeng, Chenren Xu, et al. A general and efficient approach to verifying traffic load properties under arbitrary k failures. In *SIGCOMM '24*, pages 228–243. ACM, 2024.

[25] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *SOSP '17*, pages 599–613. ACM, 2017.

[26] Xu Liu, Peng Zhang, Anubhavnidhi Abhashkumar, Jiawei Chen, and Weirong Jiang. Automatic configuration repair. In *HotNets '24*, pages 213–220. ACM, 2024.

[27] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[28] Nuno P Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *VMCAI '19*, pages 386–408. Springer, 2019.

[29] The s^2sim demo. https://routingdiagnose.com/, 2025.

[30] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *CoNEXT '14*, pages 213–226. ACM, 2014.

[31] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *SIGCOMM '16*, pages 314–327. ACM, 2016.

[32] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *SIGCOMM '23*, pages 94–107. ACM, 2023.

[33] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. Campion: Debugging router configuration differences. In *SIGCOMM'21*, pages 748–761. ACM, 2021.

[34] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting. *IEEE/ACM Transactions on Networking*, 32:2447–2462, 2024.

[35] Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. Expresso: Comprehensively reasoning about external routes using symbolic simulation. In *SIGCOMM '24*, pages 197–212. ACM, 2024.

[36] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for {Software-Defined} networks. In *NSDI '17*, pages 719–733. USENIX, 2017.

[37] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[38] Q Xiang, C Huang, R Wen, Y Wang, X Fan, Z Liu, L Kong, D Duan, F Le, et al. Beyond a centralized verifier: Scaling data plane checking via distributed, on-device verification. In *SIGCOMM '23*, pages 152–166. ACM, 2023.

[39] Xieyang Xu, Weixin Deng, Ryan Beckett, Ratul Mahajan, and David Walker. Test coverage for network configurations. In *NSDI '23*, pages 1717–1732. USENIX, 2023.

[40] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. Relational network verification. In *SIGCOMM '24*, pages 213–227. ACM, 2024.

[41] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[42] Rulan Yang, Xing Fang, Lizhao You, Qiao Xiang, Hanyang Shao, Gao Han, Ziyi Wang, Zhihao Zhang, Jiwu Shu, and Linghe Kong. Diagnosing distributed routing configurations using sequential program analysis. In *APNet '23*, pages 34–40. ACM, 2023.

[43] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *SIGCOMM '20*, pages 599–614. ACM, 2020.

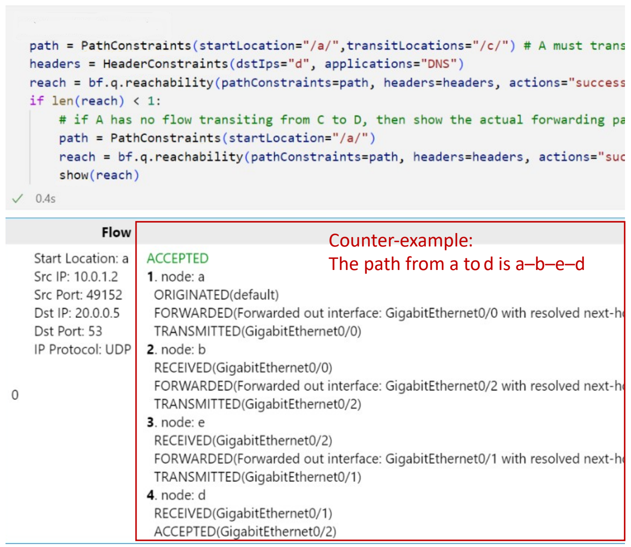
[44] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *NSDI '22*, pages 601–615. USENIX, 2022.

[45] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *SIGCOMM '22*, pages 336–349. ACM, 2022.

[46] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD '10*, pages 615–626. ACM, 2010.

A Outputs of Existing Tools

Batfish analyzed that intent 2 was violated, and returned a counter-example:



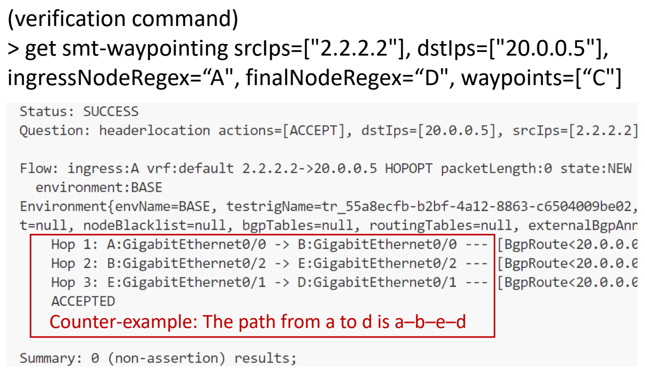
```
path = PathConstraints(startLocation="/a/",transitLocations="/c/") # A must trans
headers = HeaderConstraints(dstIps="d", applications="DNS")
reach = bf.q.reachability(pathConstraints=path, headers=headers, actions="success
if len(reach) < 1:
    # if A has no flow transiting from C to D, then show the actual forwarding pa
    path = PathConstraints(startLocation="/a/")
    reach = bf.q.reachability(pathConstraints=path, headers=headers, actions="suc
    show(reach)
✓ 0.4s
```

```
                      Flow
   Start Location: a    ACCEPTED
   Src IP: 10.0.1.2     1. node: a
   Src Port: 49152        ORIGINATED(default)
   Dst IP: 20.0.0.5       FORWARDED(Forwarded out interface: GigabitEthernet0/0 with resolved next-h
   Dst Port: 53           TRANSMITTED(GigabitEthernet0/0)
   IP Protocol: UDP     2. node: b
                          RECEIVED(GigabitEthernet0/0)
0                         FORWARDED(Forwarded out interface: GigabitEthernet0/2 with resolved next-h
                          TRANSMITTED(GigabitEthernet0/2)
                        3. node: e
                          RECEIVED(GigabitEthernet0/2)
                          FORWARDED(Forwarded out interface: GigabitEthernet0/1 with resolved next-h
                          TRANSMITTED(GigabitEthernet0/1)
                        4. node: d
                          RECEIVED(GigabitEthernet0/1)
                          ACCEPTED(GigabitEthernet0/2)
```

Counter-example:
The path from a to d is a–b–e–d

Figure 13: Batfish's output.

Minesweeper verified that intent 2 was violated, and returned a counter-example:



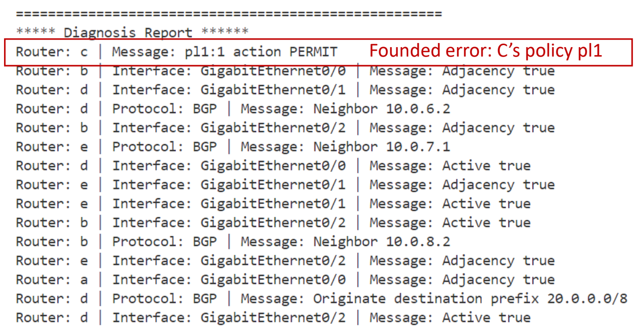
(verification command)
> get smt-waypointing srcIps=["2.2.2.2"], dstIps=["20.0.0.5"], ingressNodeRegex="A", finalNodeRegex="D", waypoints=["C"]

```
Status: SUCCESS
Question: headerlocation actions=[ACCEPT], dstIps=[20.0.0.5], srcIps=[2.2.2.2]

Flow: ingress:A vrf:default 2.2.2.2->20.0.0.5 HOPOPT packetLength:0 state:NEW
  environment:BASE
Environment{envName=BASE, testrigName=tr_55a8ecfb-b2bf-4a12-8863-c6504009be02,
t=null, nodeBlacklist=null, bgpTables=null, routingTables=null, externalBgpAnn
    Hop 1: A:GigabitEthernet0/0 -> B:GigabitEthernet0/0 --- [BgpRoute<20.0.0.0
    Hop 2: B:GigabitEthernet0/2 -> E:GigabitEthernet0/2 --- [BgpRoute<20.0.0.0
    Hop 3: E:GigabitEthernet0/1 -> D:GigabitEthernet0/1 --- [BgpRoute<20.0.0.0
    ACCEPTED

Summary: 0 (non-assertion) results;
```

Counter-example: The path from a to d is a–b–e–d

Figure 14: Minesweeper's output.

CEL verified that intent 2 was violated, but only found one error out of two:

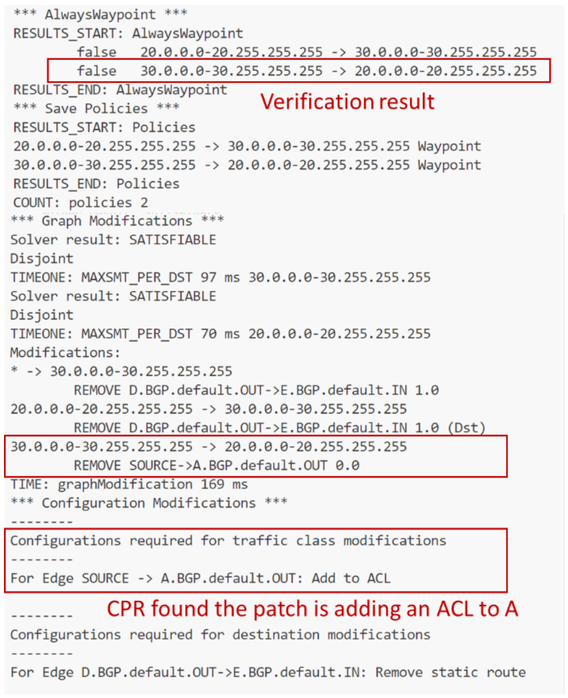


```
=====================================================
***** Diagnosis Report ******
Router: c | Message: pl1:1 action PERMIT
Router: b | Interface: GigabitEthernet0/0 | Message: Adjacency true
Router: d | Interface: GigabitEthernet0/1 | Message: Adjacency true
Router: d | Protocol: BGP | Message: Neighbor 10.0.6.2
Router: b | Interface: GigabitEthernet0/2 | Message: Adjacency true
Router: e | Protocol: BGP | Message: Neighbor 10.0.7.1
Router: d | Interface: GigabitEthernet0/0 | Message: Active true
Router: e | Interface: GigabitEthernet0/1 | Message: Adjacency true
Router: e | Interface: GigabitEthernet0/1 | Message: Active true
Router: b | Interface: GigabitEthernet0/2 | Message: Active true
Router: b | Protocol: BGP | Message: Neighbor 10.0.8.2
Router: e | Interface: GigabitEthernet0/2 | Message: Adjacency true
Router: a | Interface: GigabitEthernet0/0 | Message: Adjacency true
Router: d | Protocol: BGP | Message: Originate destination prefix 20.0.0.0/8
Router: d | Interface: GigabitEthernet0/2 | Message: Active true
```

Founded error: C's policy pl1

Figure 15: CEL's output.

CPR verified that intent 2 was violated, but found a patch that cannot fix any error:



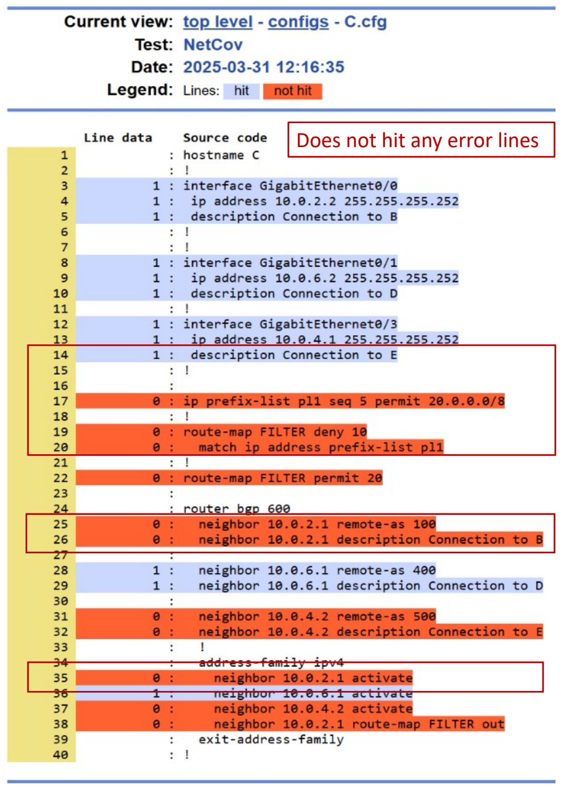
```
*** AlwaysWaypoint ***
RESULTS_START: AlwaysWaypoint
        false   20.0.0.0-20.255.255.255 -> 30.0.0.0-30.255.255.255
        false   30.0.0.0-30.255.255.255 -> 20.0.0.0-20.255.255.255
RESULTS_END: AlwaysWaypoint
*** Save Policies ***
RESULTS_START: Policies
20.0.0.0-20.255.255.255 -> 30.0.0.0-30.255.255.255 Waypoint
30.0.0.0-30.255.255.255 -> 20.0.0.0-20.255.255.255 Waypoint
RESULTS_END: Policies
COUNT: policies 2
*** Graph Modifications ***
Solver result: SATISFIABLE
Disjoint
TIMEONE: MAXSMT_PER_DST 97 ms 30.0.0.0-30.255.255.255
Solver result: SATISFIABLE
Disjoint
TIMEONE: MAXSMT_PER_DST 70 ms 20.0.0.0-20.255.255.255
Modifications:
* -> 30.0.0.0-30.255.255.255
        REMOVE D.BGP.default.OUT->E.BGP.default.IN 1.0
20.0.0.0-20.255.255.255 -> 30.0.0.0-30.255.255.255
        REMOVE D.BGP.default.OUT->E.BGP.default.IN 1.0 (Dst)
30.0.0.0-30.255.255.255 -> 20.0.0.0-20.255.255.255
        REMOVE SOURCE->A.BGP.default.OUT 0.0
TIME: graphModification 169 ms
*** Configuration Modifications ***
--------
Configurations required for traffic class modifications
--------
For Edge SOURCE -> A.BGP.default.OUT: Add to ACL

--------
Configurations required for destination modifications
--------
For Edge D.BGP.default.OUT->E.BGP.default.IN: Remove static route
```

Verification result

CPR found the patch is adding an ACL to A

Figure 16: CPR's output.

ACR depends on NetCov to identify suspicious lines first, but NetCov misses all erroneous for intent 2:



Current view: top level - configs - C.cfg
Test: NetCov
Date: 2025-03-31 12:16:35
Legend: Lines: 🟦 hit 🟥 not hit

```
      Line data     Source code
   1              : hostname C
   2              : !
   3            1 : interface GigabitEthernet0/0
   4            1 :  ip address 10.0.2.2 255.255.255.252
   5            1 :  description Connection to B
   6              : !
   7              : !
   8            1 : interface GigabitEthernet0/1
   9            1 :  ip address 10.0.6.2 255.255.255.252
  10            1 :  description Connection to D
  11              : !
  12            1 : interface GigabitEthernet0/3
  13            1 :  ip address 10.0.4.1 255.255.255.252
  14            1 :  description Connection to E
  15              : !
  16              :
  17            0 : ip prefix-list pl1 seq 5 permit 20.0.0.0/8
  18              : !
  19            0 : route-map FILTER deny 10
  20            0 :   match ip address prefix-list pl1
  21              : !
  22            0 : route-map FILTER permit 20
  23              :
  24              : router bgp 600
  25            0 :   neighbor 10.0.2.1 remote-as 100
  26            0 :   neighbor 10.0.2.1 description Connection to B
  27              :
  28            1 :   neighbor 10.0.6.1 remote-as 400
  29            1 :   neighbor 10.0.6.1 description Connection to D
  30              :
  31            0 :   neighbor 10.0.4.2 remote-as 500
  32            0 :   neighbor 10.0.4.2 description Connection to E
  33              :   !
  34              :   address-family ipv4
  35            0 :     neighbor 10.0.2.1 activate
  36            1 :     neighbor 10.0.6.1 activate
  37            0 :     neighbor 10.0.4.2 activate
  38            0 :     neighbor 10.0.2.1 route-map FILTER out
  39              :   exit-address-family
  40              : !
```

Does not hit any error lines

Figure 17: NetCov's output.

B Repair Templates of Contracts

We now present the repair templates for each contract in Table 1, illustrated using Cisco device configuration syntax. For configurations from other vendors, manual translation or adaptation of the configuration may be necessary. Note that the repair templates are not unique, and operators can tailor them according to their operational preferences. For example, operators may prefer to use the peer group to organize neighbors, or modify attributes other than local-preference to adjust route selection and enforce desired priorities.

Each template consists of several configuration lines and contains parameter holes that must be completed according to the corresponding contract(s). Specifically, holes marked with "[]" can be directly filled using contract parameters (*e.g.*, the prefix of a route), whereas holes marked with "()" require resolution via constraint programming.

In the templates, lines marked with "+" indicate configurations that must be added or updated, whereas unmarked lines serve only as contextual anchors to indicate where the changes should be applied.

isPeered contract. To repair a *isPeered(u,v)* contract, the peer configurations on both routers (u and v) must be updated. The figure below shows the repair template on u, and the same applies to the other side.

The template consists of the minimal configuration necessary to establish a BGP peering session between two routers. It specifies the remote AS number (ASN_u), the update-source interface (IP_u, usually the loopback interface), and the eBGP multihop parameter (HOP-CNT) when two routers establish a peering using non-adjacent interface IPs, followed by activation under the appropriate address family.

```
router bgp [ASNu]
+  neighbor [IPv] remote-as [ASNv]
+  neighbor [IPv] update-source [IPu]
+  neighbor [IPv] ebgp-multihop [HOP-CNT]
  address-family ipv4
+  neighbor [IPv] activate
```

isEnabled contract. Repairing a *isEnabled(u,v)* contract requires enabling the routing process on the corresponding adjacent interface for both sides. The figure below shows the repair template on router u. The procedure differs depending on the protocol: for OSPF (left), it involves advertising the adjacent interface IP (IP_u) to the corresponding OSPF process ($OSPF\text{-}ID_u$), whereas for IS-IS (right), it requires enabling the IS-IS process ($ISIS\text{-}ID_u$) on the adjacent interface (INF_u).

router ospf [OSPF-IDu] + network [IPu] area 0	interface [INFu] + ip router isis [ISIS-IDu]

isPreferred contract. To repair *isPreferred(u, r, r')* contracts, we leverage the specific mechanisms of each routing protocol. In BGP, we utilize its fine-grained policy control to adjust the local preference of the non-preferred route r', ensuring that r is selected. The figure below shows the template on router u.

Specifically, we first find the policy (RM) that router u used to import routes from v (*i.e.*, the sender of the non-preferred route r'). If such a policy does not exist in the original configuration, we create one. Then, we employ a prefix list (PFL), a community list (CML), and an as-path list (APL) that precisely match the prefix, community, and AS-path of the non-preferred route r'. Finally, we compute the concrete value of the match action (ACTION), sequence number (SEQ), and the local preference (LP), to ensure that r' is less preferred than r.

```
+ route-map [RM] (ACTION) (SEQ)
+   match ip address prefix-list [PFL]
+   match community [CML]
+   match as-path [APL]
+   set local-preference (LP)

+ ip as-path access-list [APL] permit [AS-PATH]
+ ip prefix-list [PFL] seq 1 permit [PREFIX]
+ ip community-list [CML] permit [COMMUNITY]

router bgp [ASNu]
  address-family ipv4
+  neighbor [IPv] route-map [RM] in
```

In link-state protocols such as OSPF and IS-IS, we repair this contract by recomputing the costs of all links to enforce the desired route preference using constraint programming. The figure below shows the template we used to adjust the link cost in a router.

interface [INFu] + ip ospf cost (VAR)	interface [INFu] + isis metric (VAR)

isEqPreferred contract. Repairing $isEqPreferred(u,r,r')$ contracts requires not only adjusting router configurations to assign equal preference to the routes (using the template of $isPreferred$ contracts), but also enabling the multipath policy on the corresponding routers. The figure below shows the template that enables a router to utilize up to PATH-NUM paths for traffic forwarding, where the PATH-NUM is compliant with user intents.

```
router bgp [ASNu]
+ maximum-paths (PATH-NUM)
```

isExport/isImport contract. Similar to repairing $isPreferred$ contracts in BGP, we leverage BGP's

Networks	Name	#Node	#Total Lines	Injected Error Type	#Intents [RCH (K=0)/RCH (K=1)/WPT]
WAN (Topology Zoo)	Arnes	34	3.3K	1-1, 2-1, 2-3, 3-2	10 / 10 / 2
	Bics	35	3.3K	1-1, 2-1, 2-3, 3-2	10 / 10 / 2
	Columbus	70	6.3K	1-1, 2-1, 2-3, 3-2	10 / 10 / 2
	Colt	155	13.4K	1-1, 2-1, 2-3, 3-2	10 / 10 / 2
	GtsCe	149	13.3K	1-1, 2-1, 2-3, 3-2	10 / 10 / 2
IPRAN	IPRAN-1K	1006	176.1K	1-1, 2-1, 3-1	5 / - / -
	IPRAN-2K	2006	353.2K	1-2, 2-1, 3-2	5 / - / -
	IPRAN-3K	3006	561.7K	1-1, 2-3, 3-2	5 / - / -
Fat-tree	Fat-tree4	20	0.4K	1-1, 1-2	2 / 2 / -
	Fat-tree8	80	2.7K	1-1, 1-2	2 / 2 / -
	Fat-tree12	180	8.3K	1-1, 3-2	2 / 2 / -
	Fat-tree16	320	18.8K	1-1, 3-2	2 / 2 / -
	Fat-tree20	500	35.8K	1-1, 3-2	2 / 2 / -
	Fat-tree24	720	60.7K	1-2, 3-2	2 / 2 / -
	Fat-tree28	980	95.2K	1-2, 3-2	2 / 2 / -
	Fat-tree32	1280	140.8K	1-2, 3-2	2 / 2 / -

Table 4: Detailed statistics of synthetic configurations.

fine-grained policy control to export or import a route. The only difference is that, unlike in *isPreferred*, we do not modify the preference of the exported or imported route, as shown in the figure below. The DIRECTION value (in or out) is determined by the contract type.

```
+ route-map [RM] (ACTION) (SEQ)
+   match ip address prefix-list [PFL]
+   match community [CML]
+   match as-path [APL]

+ ip as-path access-list [APL] permit [AS-PATH]
+ ip prefix-list [PFL] seq 1 permit [PREFIX]
+ ip community-list [CML] permit [COMMUNITY]

router bgp [ASN_u]
  address-family ipv4
+  neighbor [IP_v] route-map [RM] [DIRECTION]
```

isForwardedIn/isForwardedOut contract. Given an *isForwardedIn*(u, p, v) contract, we repair it by adding/updating the access control list according to the packet p. As shown in the figure below, we first locate the ACL on the adjacent interface through which router u imports packets from v. If no such ACL exists in the original configuration, we create a new one. Next, we insert a rule that exactly matches the prefix of packet p before the existing rules of the ACL. Finally, we determine the permit/deny action of the rule based on the value of the contract.

```
+ access-list [ACL] (VAR) [PREFIX]
interface [INF_u]
+  ip access-group [ACL] in/out
```

C Experimental Setup Details

Details of the synthesized networks for different network scales, injected errors, and intents are summarized in Table 4.