Astragalus: Automatic Configuration Repair for Production Networks

Zhenrong Gu
Xi'an Jiaotong University
Xi'an, Shaanxi, China

Peng Zhang
Xi'an Jiaotong University
Xi'an, Shaanxi, China

Xing Feng
Xi'an Jiaotong University
Xi'an, Shaanxi, China

Xu Liu
Xi'an Jiaotong University
Xi'an, Shaanxi, China

Abstract

Network configurations are prone to errors, which can lead to catastrophic service outages. A tool that can achieve automatic configuration repair (ACR) is highly desired by operators. Existing tools for ACR follow a *semantics-driven approach*: they model network semantics as a set of SMT constraints, and solve them for a location or fix of the error. Due to the complex semantics of networks, constructing and solving these constraints can be prohibitively expensive, making these tools neither general nor scalable. Inspired by automatic program repair (APR), we explore another direction, i.e., a *syntax-driven approach*, which generates and validates syntactically-valid candidate updates without modeling program semantics, often drawing on existing code in the same repository. Following this direction, we propose Astragalus, a syntax-driven method for ACR. It uses multiple iterations of a "localize-fix-validate" pipeline to search for repairs, and proves quite effective on configurations of our production network. Specifically, we show that Astragalus can repair every incident in multiple sizes of a synthesized network, and 97.5% of the incidents on a real network, both with 15 types of errors injected, within an average time of 6.93 seconds. It has also provided valid repairs in under 6 minutes for 7 recent network incidents or undesired changes, in a real production network with O(1,000)~O(10,000) devices.

CCS Concepts: • **Networks** → **Network reliability**; • **Software and its engineering** → *Software testing and debugging*.

Keywords: network configuration, automatic configuration repair, automatic program repair, network verification

ACM Reference Format:
Zhenrong Gu, Peng Zhang, Xing Feng, and Xu Liu. 2026. Astragalus: Automatic Configuration Repair for Production Networks. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828523.3857085



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGOPS ATC 2026, Hong Kong, Hong Kong
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857085

1 Introduction

Network outages frequently make headlines due to misconfigurations, software bugs, and hardware failures [41]. Among them, misconfiguration is a major source of network outages [26, 47]. In the network of a large cloud service provider (CSP, referred to as Provider A later), misconfigurations account for 35.4% of all incidents in the past 5 years.

To detect network misconfigurations, both the academia and industry have proposed a lot of network verifiers [1, 8, 13, 14, 19, 46, 48, 49]. These tools can check whether network forwarding behaviors align with operator intent, and conform to properties like loop freedom, and blackhole freedom.

However, these tools focus on *detecting* issues; localizing and repairing the misconfigurations is still largely left to operators. This manual process is frustrating and time-consuming, thereby delaying planned network updates; in Provider A, repairing an incident often takes far longer than detecting it (§2.1).

Similar to the Automatic Program Repair (APR) tools that greatly facilitate software development [16, 31], a tool that can achieve Automatic Configuration Repair (ACR) is highly desired by operators to help them quickly localize the root causes and resolve the incidents.

Towards ACR, there have been a handful of excellent tools [2, 17, 18], which can localize configuration errors or generate repair for misconfigurations. These tools encode network semantics, forwarding properties, and configurations as SMT constraints and solve them with off-the-shelf solvers (e.g., Z3). However, the complexity of distributed routing protocols limits both generality and scalability: constraints can only cover a subset of protocol features, and their count explodes with network size. For a 45-node fat-tree running BGP, there are already 7900 SMT constraints—prohibitively expensive for solvers to handle in reasonable time.

For inspiration, we turn to APR, which has two broad classes of approaches: *semantics-driven* approaches that solve SMT constraints for a correct-by-construction repair, and

syntax-driven approaches that *generate* and *validate* candidate updates without modeling program semantics (detailed in §2.3). We observe that existing ACR methods, e.g., [2, 17, 18], all fall within the first category of *"semantics-driven" approaches*, and thus inherit its scalability and generality limitations.

In this paper, we explore the "syntax-driven" approach and find it promising for ACR for two reasons. First, the "plastic surgery hypothesis" holds for network configurations: devices of the same role carry similar configurations, providing the raw material for syntactically-valid fixes. Second, unlike APR where validation dominates runtime due to recompilation [39], network verifiers are fast enough to evaluate many candidate fixes efficiently.

Following the direction of syntax-driven approach, we propose Astragalus, a general and scalable tool that can achieve ACR for large-scale production networks. Astragalus consists of three steps: *(1) Localization.* We localize suspicious configuration lines with techniques from APR, Spectrum-Based Fault Localization (SBFL)[4, 21]. The intuition for these techniques is that faulty lines contribute more to property violations than correct ones. *(2) Fix Generation.* According to the suspiciousness of configuration lines, we generate a set of candidate fixes including remove, insert, and modify, based on existing configuration in the same network. *(3) Validation.* We use off-the-shelf configuration verifiers to check whether the update resolves the violation without any side effects.

In summary, our contribution includes:

- We motivate a shift from semantics-driven paradigm to a syntax-driven paradigm for ACR, based on some insights from program repair.
- We design and implement Astragalus to realize ACR, a syntax-driven approach that is more scalable and general than existing methods based on SMT models.
- We test Astragalus on both real faults and synthesized network configurations with 15 types of injected errors, and show that it can repair all synthesized errors of O(100) real devices with an average time of 6.93 seconds.

2 Motivation

In this section, we first motivate the need for automatic configuration repair with real-world operational experiences (§2.1). We then show that, despite a handful of prior efforts, existing approaches all fall into the *semantics-driven* paradigm, which is correct but neither scalable nor general (§2.2). Finally, we draw inspiration from automatic program repair (APR), whose *syntax-driven* paradigm motivates our approach (§2.3).

2.1 The Need for Automatic Configuration Repair

Network verification has matured: modern verifiers like [14] can check a large network against operator intent within minutes and are now routinely deployed in production. As a result, *detecting* a misconfiguration is rarely the bottleneck, but *repairing* it is. In Provider A, verification mostly finishes within 3 minutes, yet operators still take more than 10 minutes to localize and repair 45.8% of incidents, with the worst case taking 5 hours.

Manual repair is slow because the causal chain from a configuration line to an intent violation is hard to trace. Large networks run multiple routing protocols that interact over several rounds to compute routes; devices come from different vendors, each realizing features with vendor-specific behaviors; and multiple generations of network architecture often coexist, glued together by ad-hoc configurations to interoperate [15]. Consequently, the relationship between a misconfiguration and the intent it violates is far from obvious, and even an experienced operator struggles to pinpoint the offending lines and craft a fix that restores intent without introducing new violations.

This is the gap that motivates *automatic configuration repair* (ACR): a tool that automatically localizes the root cause and produces a fix, much as Automatic Program Repair (APR) tools facilitate software development [16, 31].

2.2 Limitations of Existing Approaches

Towards ACR, there have been a handful of excellent tools that can localize configuration errors or generate repairs for misconfigurations. Although they differ in goals and techniques, we observe that they all follow a *semantics-driven* (a.k.a. correct-by-construction) recipe: they build a formal model of the network control plane and encode the repair problem as a set of SMT constraints, which an off-the-shelf solver (e.g., Z3) then solves.

CPR [17] and AED [2] encode the intent (e.g., blackhole-freedom) and the semantics of routing protocols as a set of SMT constraints, and introduce free variables for each line to represent candidate modifications. An assignment of these free variables that makes the constraints satisfied is a repair. While this SMT-based approach guarantees correct updates, it is hard to scale to large networks, due to the vast search space generated by SMT constraints and free variables. Meanwhile, the repairs tend to focus on "the most straightforward modifications" that make the constraints satisfiable, which do not necessarily correspond to a true root-cause repair. For example, on our dataset with injected misconfigurations, AED produced a repair that restores reachability by adding a static route. Although this kind of "workaround" is functionally effective, it neither resolves the underlying configuration error nor provides a clear explanation of the root cause.

CEL [18] extends Minesweeper [8] by decoupling configuration from the control logic and encoding the network model and intent using SMT constraints. It localizes misconfigurations by solving for a Minimum Correction Set (MCS) of constraints, which maps back to the root-cause

configuration statements. However, since it targets fault localization, it cannot produce an actionable repair, and its scalability is similarly limited by constraint complexity and solver performance in large networks.

The root of these limitations is that network semantics, especially distributed routing protocols, are very complex. As a result, the constraints can only cover a subset of protocols or features. Worse, the number of constraints explodes with the network size and configuration complexity: for a fat-tree network of 45 nodes running BGP, the number of SMT constraints is already as large as 7900, which makes it prohibitively expensive for SMT solvers to solve in a reasonable time.

2.3 Inspiration from Automatic Program Repair

APR has been extensively studied, and its methods are broadly classified into two categories. **Semantics-driven** approaches, also known as correct-by-construction approaches, formally encode the program repair problem as a satisfiability problem and solve it using off-the-shelf SMT solvers [29, 32]. They guarantee the repair introduces no side effects, but scalability is limited because the entire problem can have many constraints. **Syntax-driven** approaches, also known as generate-and-validate (G&V) approaches, instead *generate* candidate updates via syntactic modifications and *validate* each against a test suite, without modeling program semantics [24, 25, 38]. Many build on the *"plastic surgery hypothesis"* [6]: the code ingredients that can repair the software already exist somewhere in the codebase. This makes G&V more scalable by sidestepping expensive global semantic reasoning, and more general since no precise semantic model is needed.

From APR to ACR. ACR and APR share a highly similar problem structure: both start from an erroneous behavior and apply small, safe changes, filtered and validated against a consistent statement of intent (e.g. test cases), to drive the system from an incorrect state to one that satisfies the expected behavior. The analogy runs deeper at the artifact level: a routing configuration, especially the route policies that steer routing protocols, reads much like a *declarative program* – a set of match-action rules that the control plane evaluates to produce routes. This parallel suggests that the syntax-driven paradigm, which most existing ACR tools have overlooked, is worth exploring for configuration repair.

Unique challenges. Network configuration repair is nonetheless not a direct transplant of APR, because configuration differs from a conventional program in two key ways. First, its logic is not composed in the configuration alone but emerges from configuration coupled with protocol semantics and topology, where routes are computed by distributed protocols interacting over multiple rounds across devices, so modifying a single statement may affect the entire network, and a repair cannot be validated locally on a single device. Second, the intent against which a repair is validated is often only vaguely defined: unlike the concrete test suite available in APR, operator intent is frequently implicit or partially specified, making it harder to state precisely.

3 Overview

3.1 Key Ideas

The key idea of this paper is: instead of following the semantics-driven approach, we explore the syntax-driven approach to achieve ACR for better scalability and generality. This feasibility of applying syntax-driven approaches is due to the following two observations.

The "plastic surgery hypothesis" holds true for network configurations. As observed in Provider A's network, the devices are organized into availability zones, planes and pods; each of the zones also has multiple devices with the same role (i.e. core, aggregation, edge, etc.) to serve the purpose of redundancy. Devices with the same role share similar configurations to achieve the same functionality [12, 36]. Such configuration similarity is prominent even though Provider A's network is rather heterogeneous, spanning many geographic locations, multiple generations of architecture, and devices from multiple vendors. Similar patterns have also been observed in other real networks [22, 37]. Therefore, the ingredients needed to generate syntactically-valid candidate fixes already exist in the network. In practice, all seven real-world incidents we encountered can be repaired this way: each fix touches at most three configuration lines, with the necessary values already present in same-role devices, evidence that the ingredients needed for the repair already exist within the network (§6.1 and §6.2 give two examples).

Validation is efficient for network configurations. Network verification is well-developed [48, 49] and can quickly check configuration correctness for large networks within seconds. Provider A has an in-house control plane simulator, which can finish the computation of the data plane for a production network with 2000 nodes, within 5 seconds, enabling us to efficiently try a lot of candidate fixes. Note that in APR, the validation process dominates runtime due to the time required for recompiling and rerunning parts or the entire program [39]. Thus, the efficiency of validation implies the efficiency of repair for network configurations.

3.2 Workflow

Figure 1 shows a simplified workflow of Astragalus. First, Astragalus does a full validation of the network to identify the incident and its impact range. After that, Astragalus tries to fix the network through multiple iterations of three steps: localization, fix generation, and validation.

Step 1: Localization (§4.2). The localizer aims to identify the suspicious configurations that would cause the incident.

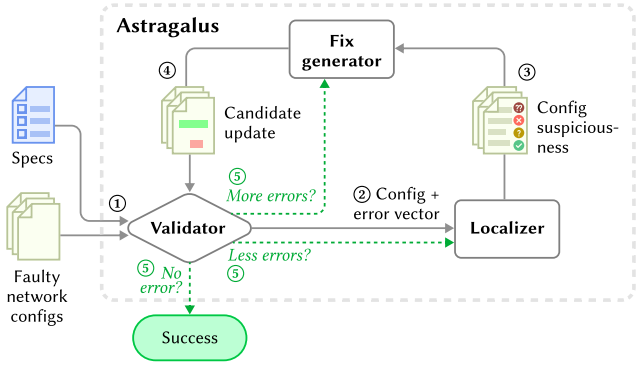


Figure 1. The workflow of Astragalus.

It tries to localize the fault based on the output from the validator. With no extra information attached, it does the localization by best effort: the identified configurations by the localizer are usually *suspicious configurations* rather than the *root causes*. After the localization, the fix generator receives the suspiciousness of each part of the configuration.

Step 2: Fix generation (§4.3). Based on the suspiciousness provided by the localizer, the fix generator generates *candidate updates* heuristically, aiming to fix the suspicious config to resolve the incident. Like the G&V approach in APR, the fix generator selects one of the suspicious configuration and applies *change operators* to produce a new configuration, i.e. the candidate update. The change operators are purely syntactic, such as removing nodes in the abstract syntax tree (AST) of the configuration [7], modifying a node or inserting a new node with values from other ASTs.

Step 3: Validation (§4.4). After applying the change operator, the validator checks if the update satisfies the specification. Since network verifiers [14, 48] can simulate the control plane from configurations, we can use off-the-shelf tools to validate candidate updates.

If the validator shows the network is fully fixed, it returns success. Otherwise, if it shows the network has fewer errors than before, it localizes again based on the new configuration (i.e., with the candidate update) to find new suspicious configurations; if it shows the network is not improved, it tells the fix generator to try again by generating a new candidate update and validate it.

The above process iterates for several rounds until a repair is found, or a maximum number of iterations is reached.

Note that Astragalus can be applied either *reactively*, to repair the network after an incident is detected, or *proactively*, to repair a proposed configuration update that fails validation before it is deployed to the network.

3.3 Merits of Astragalus

The localize-fix-validate framework has three merits, compared with semantic-based approaches.

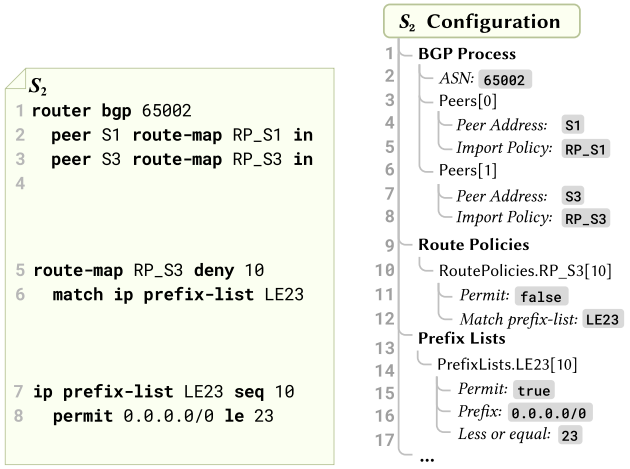
- **Scalability.** The search space for fix is relatively small, as it only involves a small subset of configurations related to the suspicious one.
- **Generality.** The framework generalizes to different protocols and vendors, since it operates upon configuration syntax, rather than semantic.
- **Flexibility.** The framework allows a flexible choice of various techniques for each stage. For example, many SBFL methods can be applied for localization.

4 Design

This section presents the design of Astragalus, which consists of three steps, i.e., localization, fix generation, and validation.

4.1 Modeling the configuration

As a syntax-driven approach to ACR, the prerequisite for Astragalus to generate a valid fix is to keep the fix syntactically correct. This section focuses on the modeling of the configuration used by Astragalus.



(a) Configuration (b) Configuration AST

Figure 2. A sample configuration snippet (a), the corresponding AST (b).

Like most routing simulators [14], Astragalus represents configurations as a vendor-agnostic AST, into which the configurations of all vendors are parsed. Figure 2(b) shows the AST for Figure 2(a): the tree organizes the configuration into BGP-process, route-policy, and prefix-list nodes, with leaves being atomic parameters (strings, IP addresses). AST nodes are typed to enforce syntactic validity (e.g., a route policy node cannot be a child of a prefix list), and only some nodes support adding or removing children of their own type.

Astragalus generates syntactically valid candidate updates on the AST. To respect these restrictions, we define:

1. A **configuration unit** is any node in the AST (used interchangeably with "AST node" hereafter).

2. A **container configuration unit** is a node that supports adding or removing children of the same type, such as BGP peers, route policies, and prefix list rules.

With the terms and configuration model being defined, we will walk through the three key steps of Astragalus in the following sections by demonstrating a repair example. Figure 3 demonstrates a typical configuration failure scenario. In this topology, S_3 is configured with three BGP aggregate addresses 20.7.0.0/23, 30.5.0.0/23, and 50.1.0.0/24. S_0 (the configuration is not shown) has an advertised network of 10.0.0.0/26. Ideally, S_2 should propagate all three aggregate addresses from S_3 to S_1, but due to a misconfiguration in the import policy from S_3 (in line 9, it should `permit` rather than deny the routes), S_2 no longer receives the route, thus causing a black hole on prefix 20.7.0.0/23 and 30.5.0.0/23.

4.2 Localization

In APR, spectrum-based faults localization (SBFL) is a widely-used technique which generates suspiciousness scores for each line of a given program. Specifically, it uses a test suite [3], and generates an *execution profile*, which contains:

1. A *coverage report*: what test cases are executed by each line of code.
2. An *error vector*: the pass / fail result for each test case.

Astragalus uses SBFL in the localization step. The following section introduces how to construct the test suite, how to execute the test suite to form the execution profile, and how to compute suspiciousness of each configuration unit.

Prefixes as the test suite. Viewing the control plane as a program, its inputs are prefixes announced inside the network (e.g., via `bgp network`)[1] or advertised by external neighbors. Unlike APR where the input space is vast, a network's prefix set is fully enumerable from device configurations. Astragalus therefore uses all `bgp network` prefixes as the test suite, supplemented by any relevant external route prefixes.

Computing the configuration coverage report. Compared with APR, computing how each configuration unit is covered by a prefix seems to be tricky, since the routing and forwarding behaviors are jointly determined by the configurations, the control plane semantics, and the routing software / simulator implementation.

However, we can still see the configurations as executable "code": When the routing software / simulator references one configuration unit while it is dealing with some route, we say the configuration unit is "executed", and the unit is considered covered by the prefix of the route.

For example, in Figure 3, propagating aggregate route 20.7.0.0/23 covers configuration units in sequence as the route travels: line 14 (S_3 aggregates the route) and line 13 (S_3 exports it to S_2); then, on S_2, line 6 (receiving the route), line 8 (matching the neighbor configuration for the incoming route), line 9 (applying import policy `RP_S3`), line 10 (the policy's first node processes it), and lines 11–12 (testing the match condition `LE23`).

For Astragalus, we implement the coverage test functionality as an interface of the in-house simulator. We start an iteration of data plane computation. The simulator records the coverage during its simulation process, and outputs the report to Astragalus.

The execution is initialized by simulating a single iteration of route advertisement from a converged data plane: each router sends all of its routes in the main RIB to neighbors. The insight of starting from a converged data plane is as follows: (1) the test cases (i.e. the `bgp networks`' and external routes' prefixes) are propagated to every destination as the control plane program suggests, so every logic that should be covered by the test cases is covered; (2) because of the convergence, route withdrawal in transient states will not affect the coverage. The converged data plane can be derived from the validation step of the previous ACR iteration to avoid duplicate calculation. By aligning Astragalus's AST with that of the simulator's, the coverage report can be used seamlessly by Astragalus.

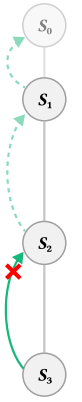
Generating the error vector. In Astragalus, we check whether a test case passes or fails, given the *specifications* of operators' intent. We use per-prefix forwarding graph as the default specification of Astragalus, as it implies many invariants in a network, such as reachability, blackhole freedom, loop freedom, route-hijacking freedom, and ECMP multipath. For each prefix in the test suite, Astragalus generates the forwarding graph of that prefix from the data plane computed by the simulator, and checks if the forwarding graph is different from the one in the specification. If the path violates the specification, Astragalus adds the prefix to the set of *failing tests*; otherwise, Astragalus adds it to the set of *passing tests*.

Astragalus also supports other specifications, as long as a specification can test whether a prefix meets its requirement. For example, all-pair reachability is a parameter-free specification which can be adapted by Astragalus; the specification reports failure if and only if any of the nodes cannot reach the owner of the prefix.

In the example of Figure 3, we mark two prefixes (10.0.0.0/26, 50.1.0.0/24) as passed, as they can be reached from all other devices; we mark the other two prefixes (20.7.0.0/23, 30.5.0.0/23) which are denied by line 9 and 10 are marked as failed.

Calculating suspiciousness scores. Astragalus can use any SBFL technique to calculate suspiciousness scores. Among the well-known methods like Tarantula [21], Ochiai [3], Jaccard [11], and D-Star[42], Astragalus uses Ochiai by default, to calculate the suspiciousness scores for configuration units. According to Ochiai, the suspiciousness of a statement s is

[1] `bgp aggregate` prefixes are excluded because they are announced only when a more-specific route exists; we trace the more-specific route instead.



	Configurations	Passing cases 10.0/26	Passing cases 50.1/24	Failing cases 20.7/23	Failing cases 30.5/23	Susp	Repair options / order
S_1							
1	`router bgp 65001`	●	●			0	
2	`  peer S0 route-map RP_S0 in`	●				0	
3	`  peer S2 route-map RP_S2 in`		●			0	
4	`route-map RP_S0 permit 10`	●				0	
5	`  set as-path rewrite 65001`	●				0	
S_2							
6	`router bgp 65002`	●	●	●	●	0.5	
7	`  peer S1 route-map RP_S1 in`	●				0	
8	`  peer S3 route-map RP_S3 in`		●	●	●	0.67	**Remove** RP_S1/② **Modify** to RP_S1/④
9	`route-map RP_S3 deny 10`		●	●	●	0.67	**Remove**/② **Insert** route-map RP_S3 allow 20/③
10	`  match ip prefix-list LE23`		●	●	●	0.67	**Remove**/② **Insert** match community-list CL1/③
11	`ip prefix-list LE23 seq 10`		●	●	●	0.67	**Remove**/②
12	`  permit 0.0.0.0/0 le 23`						**Modify** to 20.7/23/④ **Modify** to 30.5/23/④
S_3							
13	`router bgp 65003`	●	●	●	●	0.5	
14	`  aggregate 20.7.0.0/23`			●		1	**Remove**/①
15	`  aggregate 30.5.0.0/23`				●	1	**Remove**/①
16	`  aggregate 50.1.0.0/24`		●			0	

(a) Suspicious configuration units & repair options

Sort repair options

#	Repair option	Fixed?
1	**Remove** line 14	**No**
2	**Remove** line 15	**No**
3	**Remove** line 10	**No**
4	**Remove** line 9&10	**Yes**
5	**Remove** RP_S1 @ line 8	**Yes**
6	**Remove** line 11&12	**Yes**
7	**Insert** allow 20 @ line 9	**No**
8	**Insert** match community @ line 10	**No**
9	**Modify** to RP_S1 @ line 8	**Yes**
10	**Modify** to 20.7/23 @ line 12	**Improved**
11	**Modify** to 30.5/23 @ line 12	**Improved**
12	**Search exausted**	**Failure**

(b) Candidate updates

Figure 3. An example for localization and fix generation of ACR. The leftmost graph is an illustration of the topology. (a) The coverage report and suspiciousness calculation of each configuration unit in S_1, S_2 and S_3, using Tarantula; the repair options of each suspicious configuration unit. (b) The candidate repair options, and whether the repair option would fix the network. "Improved" means although the network still have issue, some of the failing prefix is solved.

calculated as:

$$\text{susp}(s) = \frac{\text{failed}(s)}{\sqrt{(\text{failed}(s) + \text{passed}(s)) \cdot \text{totalfailed}}} \quad (1)$$

where failed(s) and passed(s) represent the number of failed and passed test cases executed by the statement s, respectively; Meanwhile, totalpassed and totalfailed represent the total number of passed and failed test cases, respectively. The decision of using Ochiai SBFL technique as default is based on the evaluations in §5.6.

4.3 Generating the Fixes

The fix generation step generates candidate repairs for the suspicious configuration units computed by the localization step.

How to fix. For each suspicious configuration unit, Astragalus can use three *change operators* to fix the suspicious configuration unit.

- *Remove* deletes the node (and its subtree) of the suspicious configuration unit from the AST.
- *Insert* inserts a node into the suspicious configuration unit in the AST.
- *Modify* changes the value of the suspicious configuration unit to other values different from the current one.

Where to apply *Remove* / *Insert* / *Modify*. As §4.1 establishes, only **container configuration units** support child insertion and deletion, so *Remove* and *Insert* apply only to containers; *Modify* applies to any configuration unit.

What to *Insert* / *Modify*. As the plastic surgery hypothesis suggests, Astragalus inserts into / modifies the suspicious configuration unit with nodes that exists elsewhere. At first glance, it seems there are a large number choices for configuration lines to copy and parameters values to modify. However, we find that the number of all feasible choices is quite limited for network configurations, due to device roles and configuration syntax. More specifically, Astragalus selects a device of the *same role* (including the device itself) as the template device, and chooses a configuration unit of the *same type* as the suspicious unit. As the AST is vendor-agnostic, grafting a unit across vendors or architecture generations introduces no syntactic incompatibility. Behavioral side effects of mixing configurations from different parts of the network are left to validation (§4.4), which accepts a candidate only if all tests pass against the specification.

Figure 3(a) shows the fixes that can be generated from the suspicious configurations, only part of the *Modify* and *Insert* options are shown. For example, line 8 shows the possible repair options for configuration unit `route-map RP_S3 in`. As the import policy is optional (i.e. a container configuration unit with capacity of 1), the *Remove* option can be applied; it may also paste the route policy name `RP_S1` from line 7.

Deciding the order of fixes. Astragalus generates all possible fixes for configuration units with a suspiciousness score > 0. Then, Astragalus sorts the possible fixes from high to low based on the suspiciousness of the configuration unit from which each fix is generated. If multiple configuration units have the same score, then *Remove* updates are placed higher in the list and searched earlier, since there are at most one removing option per configuration unit, and it is more likely to yield a successful repair based on our tests; *Insert*

updates are placed after *Remove*, and finally *Modify*; if the comparison is a tie, their order in the list is determined randomly. The order is decided based on the heuristics of the data center studied. The list then forms all candidate updates of the network, as shown in Figure 3(b).

4.4 Validating the Fixes

For each fix, Astragalus first applies the fix to the configurations, thus forming a new control plane consisting of the new configurations. Then Astragalus computes the data plane, and validate the fix by testing if all test cases are passed, using the specification described by §4.2.

The above process of generate-and-validate is repeated for multiple iterations until a repair is found. In each iteration, we apply the topmost repair option in the candidate update list (as shown in Figure 3) to our network, and a new set of configurations with the repair applied (e.g. the removal of line 6 in S_2) is produced. By viewing the old and new set of configurations as two states, each iteration can be seen as transition from the old state to the new one. Therefore, Astragalus needs a search strategy to efficiently explore the state space of configurations.

Currently, Astragalus adopts a depth-first strategy as follows. After each iteration, if the new configuration state produces less errors, then we fork a new iteration based on the new state, and start searching on the state immediately. To avoid duplicates, we will check whether the state has been visited before executing the iteration. We also backtrack to the previous state if the repair option is exhausted.

If we find the state that produce no error, we report a success, with the path to the final state as the final repair.

5 Evaluation

In this section, we evaluate Astragalus to answer the following research questions: **(RQ1)** Can Astragalus accelerate the repair / fault localization of network configurations? **(RQ2)** Can Astragalus improve the scalability of ACR to larger production networks? **(RQ3)** Can Astragalus provide higher quality repairs than existing tools? **(RQ4)** Is localization (SBFL) effective for repairing network configurations?

5.1 Implementation

We implemented Astragalus with ~ 10K lines of Rust code. Astragalus is implemented with two parts:

- ACR-SDK is an SDK that provides utilities for Rust-based simulators to meet all the requirement to implement ACR on top: it provides utilities to insert hooks and generate coverage report in its simulation logic; it also provides functions for localization and fix-generation.
- Astragalus is developed as a feature of the in-house simulator that uses ACR-SDK to fully implement ACR.

5.2 Dataset and Setup

Real dataset. The real dataset is based on a subset of the production network of the CSP we studied. It is sampled based on the principle that we should be able to reproduce all historical incidents. If certain devices were taken offline due to architectural evolution, we manually selected substitute online devices with the same role. In the end, the dataset consists of 83 devices with 306 lines of configuration on average, preserving the topology, configuration, and heterogeneity (coexistence of multiple generations of architecture and devices from multiple vendors) of the production network.

Synthetic dataset[2]. We also synthesized a dataset of 5 different sizes ($k = 4, 6, 8, 10, 12$) of fat-tree topology [5] based on [35]. Each device has $97.4 \sim 163$ lines of configuration on average, respectively. To reflect the characteristics of the CSP's real data center networks (DCNs), we have modified the configurations as follows.

- In DCNs, the devices at the same layer reuse the ASNs. As a result, the AS-path of a BGP route sent across a layer twice may contain two identical ASNs. To prevent routes from being dropped because of the eBGP loop prevention mechanism, we configure `as-path rewrite` on the aggregation devices, and the corresponding route policies.
- In DCNs, devices configure BGP aggregation to reduce the total number of routes. We configure BGP aggregation on the aggregation devices, and route policies on the core devices to only import aggregate routes.
- DCNs are constantly expanding, leaving some legacy configurations in place. Even if these legacy configurations are not used, they are still kept to avoid unexpected incidents, causing redundancy in configurations. In the route policies created above, we mimic this characteristic by introducing some redundant route policies, e.g., a route can be matched by a community list, a prefix list, or both.

Despite the symmetry of fat-trees, the search space grows with the network size: on average, a fault yields 1,057, 2,115, and 3,510 candidate updates for $k = 4$, 6, and 8, i.e., the search space grows by 3.3× as the network size increases by 4× (from 20 to 80).

Misconfigurations. We consider 15 types of configuration errors, as shown in Table 1. The first 7 are real historical bugs, while the remaining 8 are bugs that we hypothesized to be at risk of occurring. For each type of error, we generate 100 instances for the above two configuration datasets. Specifically, we generate *missing* errors by randomly removing a corresponding configuration, *additional* errors by copying

[2]The synthetic dataset, including all fault instances, the results of Astragalus, and the scripts to generate and analyze them, is publicly available [20].

Table 1. Misconfigurations and support of tools.

ID	Description	Supported by		
		AED	CEL	Astragalus
1	Wrong set as-path (shorter)			✓
2	Wrong route policy application	✓	✓	✓
3	Missing prefix list rule	✓	✓	✓
4	Missing route-policy node	✓	✓	✓
5	Additional aggregate address			✓
6	Additional route policy node	✓	✓	✓
7	Additional prefix list rule	✓	✓	✓
8	Wrong route policy node permit	✓	✓	✓
9	Wrong prefix-list rule permit	✓	✓	✓
10	Wrong match-prefix	✓	✓	✓
11	Wrong route policy node priority	✓	✓	✓
12	Wrong set as-path (longer)			✓
13	Missing route policy application	✓	✓	✓
14	Missing set as-path			✓
15	Additional set as-path			✓

and pasting a corresponding configuration from a random device, and *wrong* errors by changing a parameter to some value, which can be a flip of "permit"/"deny", or a value pasted elsewhere.

These errors may lead to two types of incidents: (1) reachability incidents, where one or more devices cannot reach a prefix; and (2) path change incidents, where the forwarding paths in the misconfigured network are different from the original ones. Note that some types of the misconfigurations may not cause a reachability incident in some networks. For instance, changes in the Aspath length will not affect the network we have generated. For the real configuration dataset, 100 errors cause path change incidents for each error type. For the synthesized configuration dataset, 50 errors cause path change incidents for each error type, and 50 errors cause reachability incidents only for ER2∼ER11.

Compared methods. We compare Astragalus with two state-of-the-art methods – AED [2] which can synthesize repairs to make the configurations satisfy specified properties, and CEL [18] which can localize the configuration errors that cause property violations, but cannot return any repair. The comparison is only done for errors causing reachability incidents, as the two methods do not support specifying forwarding paths.

Specification. For the synthetic dataset, the specification is generated from the fat-tree structure; for the real dataset, it is the forwarding graph of each prefix before the misconfiguration is injected (§8).

Setup. The tests are configured with a timeout of 10000 seconds. All experiments were run on a bare-metal server, with 2x AMD EPYC 7Y83 64 core CPU running at 3.1GHz and 1007GiB of memory. Each individual repair attempt was run in parallel, with 8 logical CPU-core assigned using Linux CPU affinity.

5.3 (RQ1) Repair Performance

Table 2. Average time (in seconds) of repairing / locating errors causing two kinds of incident of AED, CEL and Astragalus in fat-tree ($k = 4$) in the public dataset. "×" means the error is not supported by the tool, and "-" marks the unavailable dataset / time.

Error ID	Reachability (s)			Path Change (s)		
	AED	CEL	Astragalus	AED	CEL	Astragalus
1	-	-	-	×	×	0.291
2	77.59	86.18	**0.019**	×	×	0.054
3	67.11	88.27	**0.016**	×	×	0.045
4	66.90	85.70	**0.028**	×	×	0.245
5	×	×	**0.065**	×	×	0.067
6	76.59	105.51	**0.017**	×	×	0.065
7	112.28	87.63	**0.014**	×	×	0.072
8	112.46	83.94	**0.027**	×	×	0.093
9	111.84	141.73	**0.016**	×	×	0.046
10	109.15	84.72	**0.018**	×	×	0.071
11	108.79	84.98	**0.013**	×	×	0.052
12	-	-	-	×	×	0.065
13	-	-	-	×	×	0.156
14	-	-	-	×	×	0.057
15	-	-	-	×	×	0.073
Average	93.63	94.30	**0.024**	-	-	0.097

Performance on the synthetic dataset. For the synthetic dataset, all three tools can correctly repair / locate 100% of the faults. Table 2 shows the time of repairing (for AED and Astragalus) / locating (for CEL) the misconfiguration that causes the error. As shown on the table, in a relatively small network, Astragalus is able to repair the fault 4∼5 orders-of-magnitude faster than the state-of-the-art solutions like AED and CEL.

Both AED and CEL cannot repair / locate Error 5, because in minesweeper, once a device is identified as the destination device, it will unconditionally advertise the dst IP to its neighbors, effectively ignoring the impact of aggregate routes; therefore, even if extra aggregate routes are injected into the edge device for ER5, they do not change the original reachability, and AED and CEL will not treat this as an error.

Performance on the real dataset. Table 3 shows the success rate and average repair time of Astragalus in the real dataset. As shown on the table, in the real dataset with more devices and configurations, Astragalus is still able to achieve an average repair time of 6.93 seconds, while most of the errors can be repaired in less than a second.

Notably, the success rate of Astragalus is no longer 100%. None of the failures is a timeout: in every failed case, Astragalus exhausts its candidate updates within the time limit. The failures happen when no other configuration in the network can be grafted to fix the misconfiguration. Reasons for that include: the misconfigured device has a role that

Table 3. Success rate and time of repairing errors in the real dataset.

Error ID	Success Rate	Median Time (s)	Average Time (s)
1	91%	0.38	1.02
2	100%	10.06	12.57
3	97%	0.23	1.63
4	98%	5.89	9.82
5	100%	0.06	7.31
6	92%	1.08	11.35
7	100%	0.14	2.31
8	100%	4.37	7.03
9	100%	0.25	0.27
10	98%	0.22	3.11
11	100%	12.35	9.87
12	100%	0.49	0.70
13	87%	0.05	16.39
14	100%	0.08	0.15
15	100%	4.75	20.41
Overall	97.53%	0.41	6.93

appears only once in the reduced topology, or the misconfigured unit is unique across the network. Such cases are rare in production networks, where devices usually have redundant counterparts of the same role.

5.4 (RQ2) Repair Scalability

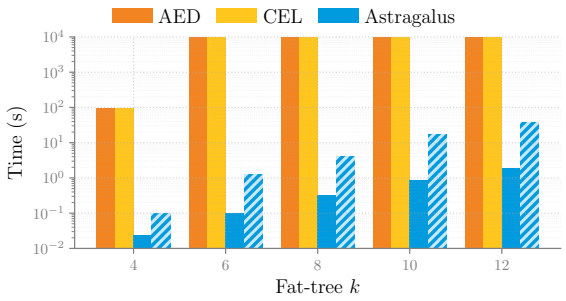


Figure 4. The average repair / locate time of AED, CEL, and Astragalus. Solid bars is for reachability incidents, and stripped bars is for path change incidents.

Figure 4 shows the repair time of Astragalus over different sizes of fat-trees in the synthetic dataset, alongside AED and CEL. The x-axis denotes the fat-tree size k (ranging from 4 to 12), while the y-axis reports repair/location time in seconds on a logarithmic scale. As evidenced in the figure, AED (orange bars) and CEL (yellow bars) exhibit severe scalability limitations. For $k = 4$, both tools require approximately 100 seconds to complete their tasks. However, for $k \geq 6$, their execution times plateau at the 10,000-second timeout, indicating they are no longer practical for moderately sized networks. In contrast, Astragalus (blue bars) demonstrates exceptional scalability, with repair times remaining orders of magnitude lower across all network sizes. As network size increases to $k = 12$, Astragalus's repair times rise gradually to less than 1 minute for path change incidents, and remain at 1.9 seconds for reachability incidents.

The difference in time of faults causing reachability incidents and path change incidents is also noticeable. As fat-tree is a robust network topology for networks with redundancy in mind, there are fewer ways to introduce a fault by altering a single configuration unit, making the error profile more monotonic. Moreover, these errors are more likely injected into the "weaker" points in the network, e.g. the edge switches' export policy to the aggregation layer. As these weak points break, the routes of the faulty prefixes are only propagated to very few devices, making the number of configuration units covered by these prefixes small; further concentrating the selection range of SBFL.

5.5 (RQ3) Repair Quality

To quantify the repair quality of Astragalus, we evaluate how many repairs are produced by directly removing the root cause of the problem, i.e. removing the misconfiguration being injected. we define *hit ratio* as the frequency of a tool giving a repair with the location *exactly the same* as the location where the misconfiguration is injected. That is, if a repair is "add a static route", it does not count as a hit since it does not localize the root cause. This metric roughly estimates the probability that the repair solutions provided by Astragalus are *directly adopted* by operators.

For each network to be repaired, CEL gives multiple *candidate predicates*, which can be seen as the possible locations of misconfigurations. To apply a fix, network engineers also need to investigate all of the candidate predicates. We calculate the frequency of the root cause as the frequency of the root cause in all candidate predicates.

Table 4. Average hit ratio of AED, CEL and Astragalus in fat-tree ($k = 4$) in the synthetic dataset. "×" means the tool cannot locate the error.

Error ID	Reachability			Path Change
	AED	CEL	Astragalus	Astragalus
1	-	-	-	40%
2	×	**50%**	12%	18%
3	×	**20%**	0%	0%
4	×	25%	**100%**	82%
5	×	×	**100%**	100%
6	×	25%	**100%**	78%
7	×	17.36%	**64%**	66%
8	×	25%	**100%**	82%
9	×	**4.68%**	0%	0%
10	×	**20%**	0%	16%
11	×	25%	**100%**	76%
12	-	-	-	0%
13	-	-	-	0%
14	-	-	-	0%
15	-	-	-	100%
All	0%	16.33%	**57.6%**	43.9%

Table 4 presents the hit ratio of AED, CEL, and Astragalus. A key note is that despite CEL's lower overall hit ratio compared to Astragalus, it includes all root causes within its candidate predicates, which means network engineers can still identify the true issue by investigating the full set of CEL's suggestions.

For all test cases, AED exclusively proposes adding static routes, which do not localize the root causes, resulting in a hit ratio of 0% across all errors. The hit ratio of Astragalus highly relies on the misconfiguration type and the priority of the candidate updates. If an easier approach (e.g. changing a node of route policy from "deny" to "permit") can fix the network, Astragalus would not search further for the root cause, e.g. adding a new route policy. This leads to the vast difference of the hit ratio: errors with easier root causes tend to have higher hit ratio.

A low *hit ratio* also does not necessarily means low fix quality. For instance, a fault introduced by removing a route policy node may be fixed by changing the route policy to another one. Since the location of the fix is different from the injected misconfiguration, it would not be counted as a "hit", but it would also reveal the root cause of the fault to the network operators.

5.6 (RQ4) Localization quality

Astragalus searches candidate updates guided by the suspiciousness given by SBFL. Without an efficient localization strategy, Astragalus may end up with randomly searching and validating in space of all configuration units, which is extremely time consuming. As a result, the localization quality directly determines the efficiency and quality of Astragalus.

We define *relative root cause location* L_r as the ratio of position of the root cause configuration in the list to the number of all suspicious configuration units. For example, if the suspiciousness of the root cause is 0.5, and there are 5 configuration units with suspiciousness higher than 0.5 out of a total of 100 configuration units, the relative root cause position is $5/100 = 0.05$.

The distribution of L_r indicates how effective the localization is. If we choose the fix randomly from all suspicious configuration units, L_r should satisfy uniform distribution, where the root cause should be found anywhere with equal possibility density; if the localization method is effective, we should expect more root causes to be found in prior positions.

Figure 5 shows the cumulative distribution function (CDF) of the relative root cause location L_r for 4 SBFL techniques: Ochiai, tarantula, jaccard, and D-Star (with parameter 2). In both path change and reachability test cases, Ochiai, Jaccard and D-Star^2 do not have observable difference in the CDF; but all of the three methods can find the root causes faster than Tarantula.

We choose Ochiai as the default because it is one of the three faster SBFL techniques, but using Jaccard and D-Star^2

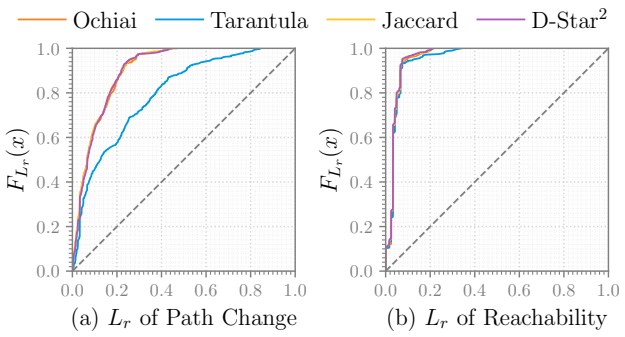


Figure 5. CDF of *relative root cause location* (L_r) of the 4 SBFL techniques. The diagonal line indicates L_r of selecting suspicious configurations randomly.

will not make localization observably less accurate. Choosing Tarantula, however, would make the localization, and the repair slower. In fact, in fat-tree with k as small as 6, Tarantula cannot find a valid repair within 6000 iterations for 9 cases of path change incidents, making the repairing time more than 5 minutes; while using Ochiai, all faults in the synthesized dataset can be repaired within 2607 iterations. This number also reveals a long tail: even with the best SBFL technique, the hardest fault takes up to 2607 iterations to repair. This is affordable in the networks we evaluate, but may limit repair efficiency in larger networks, where each iteration is more expensive. The same tail appears in the real dataset as the gap between the median and average repair time in Table 3, e.g., 0.05 s versus 16.39 s for ER13.

5.7 Microbenchmarks

To gain a deeper understanding of the characteristics of the repairing process and outcomes by Astragalus, we tested (1) the time of the three steps – localization, fix generation and validation, taken separately; (2) the proportion of three operators – *Remove*, *Insert* and *Modify*, used in the repair.

Figure 6 shows the running time of the three modules, in the synthesized dataset. We see that most time is spent on localization and validation, where the simulator involves to compute the data plane / generate the coverage report. Generating fixes, however, takes much less time since the simulator is not involved.

Figure 7 reveals the proportion of the three change operators for fixing each kind of misconfigurations. In total, although ER5, ER6, ER7 and ER15 are the only 4 out of the 15 misconfigurations whose root cause should be removed, 62.7% of the repairs use the *Remove* change operator. The proportion of *Remove* operator is even higher for the reachability incident dataset: 79% of the faults can be repaired with the *Remove* operator. In the real dataset, *Remove* is still the most dominant change operator, which can fix 37.4% of the faults. The second most used operator is *Insert*, counting up

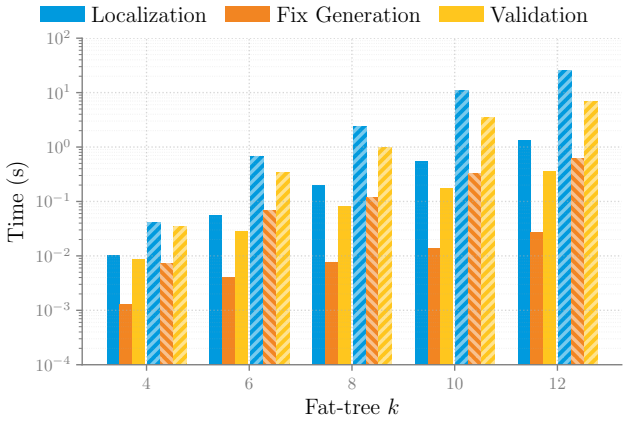


Figure 6. The average time of localization, fix generation, and validation of Astragalus, in the synthesized dataset. Solid bars is for reachability incidents, and stripped bars is for path change incidents.

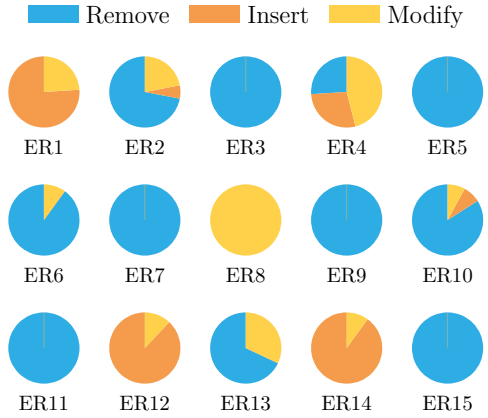


Figure 7. The proportion of three change operators applied in fixing fat-tree 8's path change incidents.

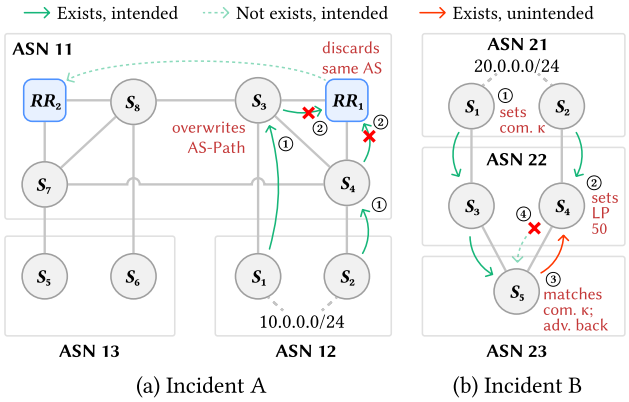
for 18.4% of the fixes. Only 15.1% of the cases are fixed by *Modify* operator.

6 Real-World Experiences

As a tool aiming for ACR in real production networks, Astragalus has been tested on a large service provider's networks, which have O(1,000) ~ O(10,000) devices. In the following, we share 2 of the incidents highlighting the scalability, repair quality, and limitations of Astragalus. Both cases below are reactive uses of Astragalus; we choose them because their root causes are known, so we can evaluate the quality of repairs.

6.1 Incident A: Blackhole due to VSBs

Network and configurations. The network comprises ~1000 devices, and each device has ~20,000 lines of configuration



(a) Incident A (b) Incident B

Figure 8. Two real-world network incidents studied. Arrows indicate route propagation, and circled numbers mark the order of events. For Incident B, only S_4's perspective is drawn to avoid clutter; the same flapping occurs symmetrically on S_3.

on average. As shown in Figure 8(a), the network spans multiple data centers (only two are shown, AS12 and AS13), connected by a transit AS (AS11). In AS11, each neighboring data center is served by two peering switches (S_3, S_4 for AS12; S_7, S_8 for AS13) and a BGP route-reflector (RR). The two RRs peer over IBGP to exchange routes between the data centers, while each peering switch establishes IBGP only with its corresponding RR ($S_3, S_4 \leftrightarrow RR_1$; $S_7, S_8 \leftrightarrow RR_2$).

Many data centers in the network reuse ASNs in some layer of the network. As a result, the AS-path of a BGP route sent from one data center to another may contain two identical ASNs. To prevent routes from being dropped by the EBGP loop prevention mechanism, the peering switches in AS11 apply an `as-path rewrite` import policy to routes from AS12 / AS13: any AS-path containing ASNs from the originating data center is replaced with ASN 11 repeated several times.

The incident. The operators found that the routes originated from ASN 12 could not be received by the devices in ASN 13. The root cause is that the operators overlooked a vendor-specific behavior (VSB) of the RRs: besides the standard EBGP loop prevention, the RRs also perform IBGP loop prevention, discarding any route whose AS-path contains their own ASN. Since S_3 and S_4 rewrote the AS-path of the AS12 route into several 11s, RR_1 (in AS11) treated it as an IBGP loop. As a result, RR_1 could not learn the routes from AS12, and all of RR_2, S_7, S_8, S_5 and S_6 could not receive it.

The repair given by Astragalus. Astragalus gave 2026 candidate updates for the network fault. Among these updates, the 23rd most suspicious update is to *Modify* `set as-path` **`replace`** to `set as-path` **`exclude`** in the import policy on S_3 for routes from S_1 (in AS12). Effectively, instead of overriding the AS-path by its own ASN, it empties the AS-path attribute directly. Thus, it stops RR_1's IBGP loop prevention

from triggering unintentionally, while still preventing route drops by EBGP loop prevention in other data centers.

Astragalus adopted this fix, which restores reachability from AS13 to AS12 via S_3. Nonetheless, AS12 is still unreachable via S_4, since S_4's route was still being rejected. Astragalus continued searching, and found the same fix for S_4 in another 23 iterations. The network is now fixed with correct ECMP paths to AS12. In total, the repair solution is found in 118.35 seconds from 47 iterations (including 1 verification iteration at the very beginning).

Lessons learned. (1) Astragalus is scalable. It can be applied on real, hyper-scale data center networks, providing repairs in minutes. (2) SBFL in network configuration is effective, as the root cause is found in top 1.1% of the candidate updates.

6.2 Incident B: Route Flips

Compared to Incident A, Incident B features a more complex and obscure phenomenon and root cause.

Network and configurations. The network is larger than Incident A's, with 10× more devices and 4× more total lines of configuration. As shown in Figure 8(b), the devices of AS21 originate a BGP aggregate `20.0.0.0/24` and advertise it into AS22, using an export policy that sets a community attribute. In AS22, the switches assign a lower local preference to BGP routes coming from AS21. Switches in AS23 re-advertise routes back into AS22 by matching community κ, which is normally set only by specific downstream devices.

The incident. After adding two new switches S_1 and S_2 in AS21, a route flapping happened between S_3, S_4 and S_5. The root cause was: the configuration of export policy in S_1 and S_2 mistakenly set the community to κ. Consider the propagation of `20.0.0.0/24` from S_1. The route propagates normally along $S_1 \rightarrow S_3 \rightarrow S_5$; but since S_5 (in AS23) matches κ in its export policy to AS22, it advertises the route back to S_4. S_4 also receives `20.0.0.0/24` directly from S_2, but because that route carries the low local preference set by the import policy from AS21, S_4 instead prefers the looped route from S_5, and withdraws the route it had advertised to S_5. The same problem happens symmetrically on S_3. But once S_4's advertisement is withdrawn, S_3 again prefers the route from S_1.

The repair given by Astragalus. In this incident, Astragalus does not fix the network by directly addressing the root cause. Instead, it applies repairs at two locations, S_3 and S_4: First, in the 5$^{\text{th}}$ iteration, Astragalus finds the candidate update on S_4 to *Modify* the import policy from AS23 from `allow` to `deny`. Since S_4 can no longer receive the route from S_5, it prefers the route from S_2 despite its low local-preference. The route from S_4 still matches community κ at S_5 and is still preferred at S_3; but since it is no longer withdrawn from S_4, the flapping stops.

The next step is to fix the incorrect route propagation from S_5 to S_3. In the 15$^{\text{th}}$ iteration, Astragalus finds the candidate update on S_3 to *Modify* the import policy from AS21 to a policy that sets the *local-preference* of routes to 200. This policy is grafted from the import policy of a device with the same role as S_3 for routes from an SDN controller in the network. As the direct cause of the incident is that S_3 and S_4's local preference of routes received from AS21 is lower, the repair candidate removes the condition for S_3 to prefer routes of `20.0.0.0/24` from S_5. Finally, the route from both S_1 and S_2 is propagated correctly. In total, the repair solution is found in 315.93 seconds from 15 iterations. Each validation, which recomputes the data plane of the entire network, takes about 8s while routes are flapping and about 3s once the flapping is resolved. As this is the largest network we deal with at Provider A, we find the validation cost acceptable at production scale.

Lessons learned. The repairs generated by Astragalus may not *be* the root cause, but they can be helpful to *find* the root cause. In this case, the lower local-preference of S_3 and S_4 is a direct cause of the route flapping problem. With the repair, network engineers can easily find the reason why routes in S_5 are sent back to S_4, further find the matching condition and the mistakenly set community value. On the other hand, one might not take the suggestion as a permanent solution for a network fault, as it might be not robust and only works for the specification when fixing. The repair also shows that ingredients can come from heterogeneous parts of the network: the grafted policy was written for an SDN controller, a different kind of peer from the switches in AS21.

7 Related Work

Configuration repair and synthesis. The closest prior work targets automatic configuration repair. CPR [17] and AED [2] synthesize repairs by encoding control-plane semantics and operator intent as SMT constraints and solving for a satisfying configuration, while CEL [18] localizes the root cause by solving for a minimum correction set. Provenance-based methods [10, 43, 44] trace abnormal forwarding events back to candidate root causes through a provenance graph. Data-driven configuration management has also been explored: Auric [28] uses recommendation techniques to automatically generate cellular network configurations. As discussed in §2.2, all repair approaches build a formal or learned model of the network and reason over it; in contrast, Astragalus adopts a *syntax-driven* approach that grafts existing configurations and validates them, making it more scalable and general.

Network verification, simulation, and misconfiguration detection. A large body of work checks configurations against intent but stops short of repairing them. Simulators such as Batfish [14] and verifiers such as Minesweeper [8],

APKeep [49], DNA [48], Tiramisu [1], VeriFlow [23], and Plankton [34], among others [13, 19, 33, 40, 46, 50], detect violations of forwarding properties. Complementary data-driven techniques detect likely misconfigurations without a formal specification: SelfStarter [22] infers configuration templates and flags outliers, Campion [37] debugs configuration differences across similar routers, and pattern-mining [9] surfaces statements that deviate from common patterns. These tools *detect* or *localize* errors but leave the fix to operators, which is precisely the manual burden that motivates Astragalus.

Automatic program repair. Astragalus adapts ideas from APR, which falls into two paradigms: semantics-driven repair that solves constraints for a correct-by-construction patch [29, 30, 32], and syntax-driven generate-and-validate repair [24, 25, 27, 38] built on the plastic surgery hypothesis [6, 45]; we refer the reader to surveys [16, 31] for a broader treatment. Our localization step builds on spectrum-based fault localization (SBFL) [3, 4, 21, 42], which scores a statement's suspiciousness from its coverage by passing and failing tests. To our knowledge, Astragalus is the first to adapt syntax-driven APR and SBFL to network configuration repair.

8 Discussion

Using LLMs for fix generation. We generate fixes by grafting existing configurations, following the plastic surgery hypothesis. Recent studies suggest that LLMs can generate better fixes [45]. However, generating a fix with an LLM is much slower than with grafting. Since Astragalus's efficiency comes from quickly trying out a lot of candidate locations that may not be the root cause, invoking an LLM at every candidate location can be quite wasteful. The cost of invoking an LLM may be amortized if it is used to prioritize candidate updates rather than to generate each one; we leave this as future work.

Repairing data plane misconfigurations. Astragalus currently repairs only misconfigurations for routing protocols (e.g., BGP). There are other configurations used for data plane forwarding (e.g., policy-based routing and NATs), and can also cause network incidents if there are errors. For data-plane configuration, the same SBFL approach also works. By viewing packets as test cases, and test packet forwarding properties (reachability, blackhole-freedom, etc.), the localization approach can be also applied on data plane configurations. As the simulator only supports simulation of the control plane, Astragalus currently is unable to fix these configurations. In the future, we will try to implement the ACR principle on data plane or general network simulators to enable the repair of data plane configurations.

Repairing wide-area network configurations. In this paper, we design and evaluate Astragalus based on data center network configurations. For other kinds of networks such as WANs, we tentatively think the plastic surgery hypothesis also holds, as there are also some level of redundancy (e.g. multiple border routers peering with the same ASes). Therefore, we hypothesize Astragalus can be a general tool for repairing these networks. We leave the evaluation of Astragalus on WANs as future work.

Repairing missing route origins. When an incident is caused by a missing route origin, e.g., a missing advertisement statement, route redistribution, or static route, the affected prefix is never generated, so its failing test covers no configuration unit and leaves SBFL nothing to localize. We leave such incidents to manual troubleshooting, as their cause is relatively obvious.

Specification. Astragalus is only as reliable as the specification it validates against. Specifying the intended forwarding graph of every prefix by hand is impractical for a real network, as operator intent is often implicit (§2.3), and how far this can be automated depends on the network. In a well-structured network such as a fat-tree, the intended forwarding follows from the topology, so the specification can be generated automatically. For irregular networks, Astragalus uses the forwarding graphs computed from the configuration *before* a change as the specification for the configuration after it, assuming the change should not alter the forwarding graphs. This applies to a much wider range of networks but is only semi-automatic: for changes intended to alter the forwarding graphs, operators need to edit the affected part of the specification manually. Our evaluation does not consider such cases, since an injected misconfiguration is never an intended change. Automatically inferring the intent behind such changes is an interesting direction for future work.

9 Conclusion

In this paper, we motivate the need of automatic configuration repair (ACR) with experiences of real-world network incidents. Inspired by APR, we propose to achieve ACR with a syntax-driven approach, which is general and scalable. Specifically, we design and implement Astragalus, and evaluate it based on configurations of real production networks. We show Astragalus can repair over 97.5% of the injected errors within an average of 6.93 seconds.
This work does not raise any ethical issues.

Acknowledgement. We thank the anonymous reviewers for their constructive feedback. We also thank our shepherd for the help throughout the preparation of this paper. This research was supported by the National Laboratory Special Program of Peng Cheng Laboratory under Grant No. PCL2026A04, and the National Natural Science Foundation of China under Grant No. 62672400 and No. 62272382. Peng Zhang is the corresponding author for this paper.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2019. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043* (2019).

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 482–495.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[4] Pragya Agarwal and Arun Prakash Agrawal. 2014. Fault-localization techniques for software systems: A literature review. *ACM SIGSOFT Software Engineering Notes* 39, 5 (2014), 1–8.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.

[6] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 155–168.

[9] Jyotirmay Chauhan, Devon Lee, Emily Yu, and Aaron Gember-Jacobson. 2021. Detecting Configuration Errors Via Pattern Mining. *Proc. of NetVerify@ lCNP, Dallas, TX, USA. IEEE* (2021).

[10] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 115–128.

[11] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 595–604.

[12] Facebook Engineering. 2014. Introducing Data Center Fabric: The Next-Generation Facebook Data Center Network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/. [Online; accessed 06-December-2026].

[13] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 217–232.

[14] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 469–483.

[15] Zhaoyu Gao, Anubhavnidhi Abhashkumar, Zhen Sun, Weirong Jiang, and Yi Wang. 2024. Crescent: Emulating Heterogeneous Production Network at Scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1045–1062.

[16] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*. 1219–1219.

[17] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 359–373.

[18] Aaron Gember-Jacobson, Ruchit Shrestha, and Xiaolin Sun. 2022. Localizing router configuration errors using minimal correction sets. *arXiv preprint arXiv:2204.10785* (2022).

[19] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 300–313.

[20] Zhenrong Gu, Peng Zhang, Xing Feng, and Xu Liu. 2026. Artifact for ATC 2026 Paper "Astragalus: Automatic Configuration Repair for Production Networks". Zenodo, version v1.0.0, https://doi.org/10.5281/zenodo.22910277. https://doi.org/10.5281/zenodo.22910277

[21] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[22] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 999–1013.

[23] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*. 49–54.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[25] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE, 282–291.

[26] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*. 29–35.

[27] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[28] Ajay Mahimkar, Ashiwan Sivakumar, Zihui Ge, Shomik Pathak, and Karunasish Biswas. 2021. Auric: Using data-driven recommendation to automatically generate cellular configuration. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 807–820.

[29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[30] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[31] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[32] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[33] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices* 51, 1 (2016), 69–83.

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

[35] Divya Raghunathan, Ryan Beckett, Aarti Gupta, and David Walker. 2022. ACORN: network control plane abstraction using route nondeterminism. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 261–272.

[36] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review* 45, 4 (2015), 183–197.

[37] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 748–761.

[38] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*. 61–72.

[39] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.

[40] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications*. 765–780.

[41] Wikipedia contributors. 2024. 2021 Facebook outage — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=2021_Facebook_outage&oldid=1221077563. [Online; accessed 18-June-2024].

[42] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 21–30.

[43] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for {Software-Defined} Networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 719–733.

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

[45] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The plastic surgery hypothesis in the era of large language models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 522–534.

[46] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. 2020. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 599–614.

[47] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 241–252.

[48] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 601–615.

[49] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. {APKeep}: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 241–255.

[50] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 336–349.