Automatic Configuration Repair

Xu Liu
Xi'an Jiaotong University

Peng Zhang
Xi'an Jiaotong University

Anubhavnidhi Abhashkumar
ByteDance

Jiawei Chen
ByteDance

Weirong Jiang
ByteDance

ABSTRACT

Networks are error-prone due to misconfigurations, and it is hard to identify the root causes in the configuration and find a repair due to the size and complexity of networks running distributed routing protocols. Thus, we advocate *Automatic Configuration Repair (ACR)* to reduce the manual effort. Specifically, we draw some insights from the field of *Automatic Software Repair (ASR)*, crystallize some lessons learned from the real-world repair experience of a large service provider, and propose some directions to realize ACR. Inspired by the generate-and-validate approach from ASR, we propose *localize-fix-validate* as a possible approach to realize ACR.

CCS CONCEPTS

• **Networks → Network reliability**.

KEYWORDS

misconfiguration, localization, repair

ACM Reference Format:
Xu Liu, Peng Zhang, Anubhavnidhi Abhashkumar, Jiawei Chen, and Weirong Jiang. 2024. Automatic Configuration Repair. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696895

1 INTRODUCTION

Network outages frequently made headlines due to misconfigurations, software bugs, and hardware failures [24]. Many of these outages last a long time, have a wide and even global impact, and lead to severe economic loss. A major source of the outages is misconfiguration. According to our study of a large service provider (ByteDance), 35.4% of incidents were caused by misconfiguration.

To detect network misconfigurations and improve network reliability, academia and industry have proposed some methods, including: *(1) Network simulation* is based on device images [11] or customized model [10], simulating the network to predict its behaviors under specific configurations and failure scenarios; *(2) Network verification* [29, 30] uses formal methods to reason about network behaviors against operator intent, such as k-failure tolerance, loop-freedom, and blackhole-freedom.

However, existing simulation or verification methods can only *detect* issues or *reduce* the impact rather than *resolve* it, i.e., localizing the misconfigurations to blame and repairing the configurations to meet the intent again. As a result, operators need to do the localization and repair manually.

Such a manual resolving process can be frustrating and time-consuming, thereby delaying the planned network updates. In ByteDance's network, operators took more than 30 minutes in 16.6% of cases to localize and repair the issue, with the longest one taking more than 5 hours. The reason for the expensive overhead is the complexity of the distributed networks. First, large enterprise networks may have multiple routing protocols that interact with each other in several rounds to compute the best routes. Moreover, the networks may have routers from different vendors, each with vendor-specific behaviors for implementing some features. Worse still, multiple generations of network architectures may coexist, each with distinct routing strategies [11]. Consequently, the causal relationship between misconfigurations and intent violations is less obvious, making it difficult for even an experienced operator to identify and resolve the root cause of an incident.

We advocate Automatic Configuration Repair (ACR) to help operators localize the root causes and find a repair. Realizing ACR is challenging in the following two aspects.

Correctness/Effectiveness. An effective repair should not only eliminate the intent violations but also guarantee not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HOTNETS '24, November 18–19, 2024, Irvine, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696895

Configs	Types	Lines	Ratio
Route	Missing redistribution of static route	M	20.8%
PBR	Missing permit rules in PBR	M	12.5%
	Extra redirect rule in PBR	S	4.2%
Peer	Missing peer group	M	16.6%
	Extra items in peer group	M	12.5%
Policy	Missing a routing policy	M	8.3%
	Fail to dis-enable route map	S	4.2%
	Override to wrong AS number	S	4.2%
	Missing items in ip prefix-list	S/M	4.2%/12.5%

Table 1: The types of misconfiguration. M=Multiple, S=Single.

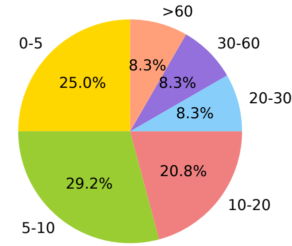


Figure 1: Resolving time (mins) of misconfiguration.

to introduce more violations. However, due to the inherent complexity of distributed networks, it is hard to reason about the impact of a repair across devices.

Performance/Scalability. An efficient repair should scale to large networks with tens of thousands of devices and finish as soon as possible. Considering that each device can have thousands of lines of configuration, any of which may be possible for generating a solution, the search space can be astronomic. Searching for a feasible repair in such a space is an NP-Complete problem [25, 26]. Moreover, it is common for a feasible repair to require the modification of multiple lines of configuration, which makes the problem even harder.

Existing tools for localizing and repairing faulty configurations include provenance and synthesis methods. In the following, we show why they cannot address the challenges.

Provenance methods [7, 25, 26] are efficient but not necessarily correct. This is because they reduce the search space to a relatively small provenance graph so they can identify the root cause by tracing the source of abnormal events. However, the identified source may not help to generate a feasible repair because it may not be the faulty one or will introduce regressions (§2.3).

Synthesis methods [1, 13] are correct but not scalable. This is because they use formal methods to completely encode the network semantics and operator intents as SMT constraints, so they can guarantee to produce a repair without side effects by systematically searching for a solution that satisfies all the constraints. However, it cannot scale to large networks, because the search space could be extremely large when the number of constraints grows with network size (§2.3).

In order to address the above two challenges, we conduct a comprehensive study of over 100 incidents of a production network, and draw some insights from the field of Automatic Software Repair (ASR) [12, 20].

Then, we propose the localize-fix-validate approach to achieve ACR. *(1) Localize.* We use Spectrum-Based Fault Localization (SBFL) [3, 16] to localize the suspicious lines of configuration. The intuition behind such a method is that faulty lines of configuration contribute more to intent violations, compared with the correct lines. *(2) Fix.* We apply *change operators* to the suspicious lines of configuration to generate a set of candidate fixes (referred to as *updates*). The change operators are pre-defined based on real-world repair experience. The intuition is that error types of misconfiguration in real-world incidents are very limited, allowing us to define a set of templates for repair. For example, in ByteDance, there are only 9 types of errors out of over 100 real-world incidents we have studied. *(3) Validate.* We use off-the-shelf network verifiers to check whether the update resolves the violation without introducing more violations. This step is efficient owing to the advance of incremental verifiers like DNA [29], which can incrementally verify the affected intents based on intermediate results without verifying all of them from scratch.

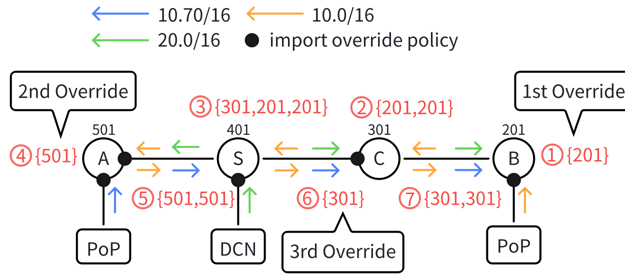
Such a localize-fix-validate approach is scalable since it does not need to explore the whole search space as synthesis approaches. Based on our experiences in ByteDance's network, we find that heuristically using pre-defined change operators strategy is also quite effective since there are only 9 types of errors out of over 100 real-world incidents.

2 MOTIVATION

2.1 Experience of Network Incidents

We investigated more than 100 incidents in ByteDance's worldwide network in detail. This section will introduce the background and motivation for proposing ACR in terms of root causes and manual effort.

Root cause. In all the incidents we studied, misconfiguration is the primary root cause, accounting for 35.4%, followed by hardware failures (34.6%), software bugs (25.3%), and vendor-specific behaviors (4.7%). Table 1 shows the types of misconfiguration. We can see that 83.2% of cases are related to multiple lines of configurations, while more than half of them are simple and related to specific IP prefixes: missing static routes (20.8%), PBR rules (12.5%), and ip-prefix list members (12.5%).



(a) The process of routing flapping.

Configuration Snippet	10.70/16	20.0/16	10.0/16	Susp
1 **bgp** 501	•	•	•	0.5
2 **peer** PoP **group** Downside	•	•	•	0.5
3 **peer** S **group** DCNSide	•	•	•	0.5
4 #				
5 **address-family Ipv4 unicast**	•	•	•	0.5
6 **peer** Downside **enable**	•	•	•	0.5
7 **peer** Downside **route-policy** Override_All **import**	•			0
8 **peer** DCNSide **enable**	•	•	•	0.5
9 **peer** DCNSide **route-policy** Override_All **import**		•	•	0.67
10 #				
11 **ip prefix-list** default_all **index** 5 permit 0.0.0.0 0	•	•	•	0.5
12 #				
13 **route-policy** Overide_All permit **node** 10	•	•	•	0.5
14 **if-match ip address prefix-list** default_all	•	•	•	0.5
15 **apply as-path** 501	•	•	•	0.5
16 #				

(b) The configuration snippet of router A.

Figure 2: The example of routing flapping.

Manual effort. Figure 1 shows the time of the manual repair process for incidents caused by misconfiguration. The time was collected by inspecting the duration between the end of mitigation and the end of the analysis, including localization and repair. We can see that 16.6% of the cases took more than 30 minutes, with the longest taking more than 5 hours. The reason for the expensive overhead is the complexity of networks using distributed routing protocols and the ineffectiveness of commonly used analyzing tools (show route table, traceroute, pingmesh, etc.)

In the last two years, the percentage of network changes pre-checked by network verifiers has grown from 20.6% to 67.1%, and most potential incidents due to misconfigurations have been captured. However, after capturing the incidents, operators still need to manually localize and repair the misconfigurations, which delay the deployment of network changes. A tool that can automatically localize and repair network misconfigurations is highly desired.

2.2 An Example Incident

Next, we use an example incident in ByteDance's network to show the limitations of existing methods.

The network. Figure 2 shows the example network running BGP, which consists of four backbone routers (A, B, C, and S), two Points of Presence (PoPs) and one Data Center Network (DCN), Figure 2b shows a configuration snippet of router A, where the as-path override policy (lines 13-16) rewrites the AS_PATH of all routes received from connected PoP and router S (lines 5-11) to its own AS number (lines 1, 15). Similar routing policies are also configured on routers B, C, and S, marked in Figure 2a.

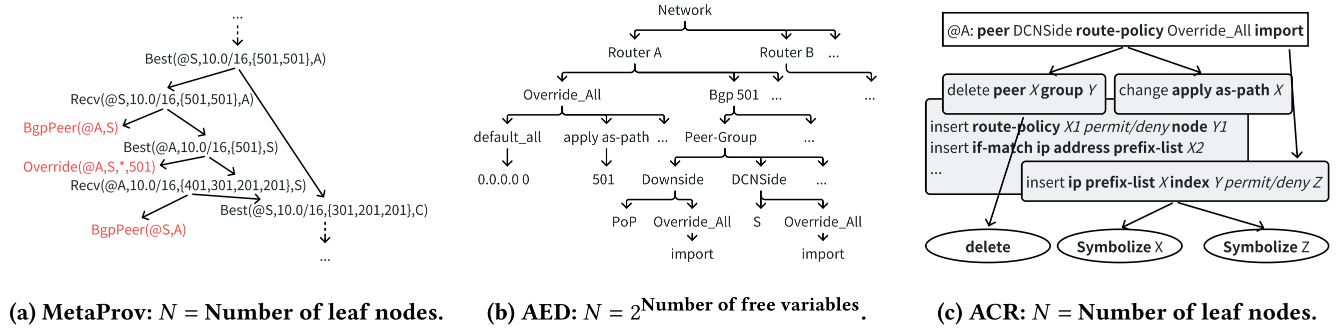
The incident. Routers S and C are not configured as BGP neighbors until a new reachability intent requires that the DCN of router S access the PoP of router B, resulting in three advertisements into the backbone, one of which caused a route flapping for prefix 10.0/16. The orange arrows in Figure 2a show the route propagation for 10.0/16. For simplicity, we only show the AS_PATH attribute of this route. Steps 1, 4, and 6 rewrite the AS_PATH as the router's AS number due to the override policy.

The root cause and repair. Operators found the root cause was the override policies on A and C, which increased the priority of the route by shortening the length of the AS_PATH. As a result, router $S(C)$ mistakenly set the next hop for 10.0/16 to $A(S)$ instead of $C(B)$, and announced this route update into the network. To repair the problem, operators changed the 0.0.0.0 0 in the ip prefix-list (e.g., line 11 in Figure 2b) to 10.70/16 and 20.0/16, which only rewrote the routes originated from the connected PoP and DCN.

2.3 Limitations of Existing Methods

Existing works try to locate or repair the root cause of configuration errors by provenance [7, 14, 25, 26] or synthesis [1, 13] methods. However, these methods either satisfy effectiveness or efficiency, but none of them satisfy both requirements.

Provenance-based methods are efficient but not necessarily correct. Specifically, they draw a directed acyclic graph to record the derivation between network events and trace the source of abnormal events in the graph to identify the root cause. Then, an update can be generated by modifying the value of the identified source. This kind of method is efficient by reducing the search space to a relatively small graph, but the update may not necessarily be correct. First, such a trace can only reveal configurations that are *relevant* to the target event rather than localize the *responsible* ones. Second, it ignores the interaction between configurations and may introduce new intent violations.



(a) MetaProv: N = Number of leaf nodes. **(b) AED: $N = 2^{\text{Number of free variables}}$.** **(c) ACR: N = Number of leaf nodes.**

Figure 3: The search space N of each method.

For example, the search space of MetaProv [25] is the set of leaf nodes of the provenance tree (Figure 3a), each of which is a predicate representing a line in the configuration. MetaProv may identify that line 11 of Figure 2b needs to be updated so that it prevents router A from rewriting routes of 10.0/16. However, the override policy on C will still rewrite the routes received from S, making C choose S as the next hop for 10.0/16. This will form a forwarding loop between C and S, because C is the next hop for 10.0/16 on S. Therefore, merely modifying the policy on A cannot resolve the problem, i.e., it is not a correct update.

To find updates with multiple changes, provenance methods can extend the search space to the power set of N, i.e., 2^N. However, such space is extremely large even in our example network, i.e., at least 2^{12} for router A, which contains 12 lines in the snippet.

Synthesis methods are correct but not scalable. Specifically, they use formal methods to model the network semantics and operator intent as a set of SMT constraints; if some intent is not satisfied, the methods introduce a lot of free variables to represent candidate repairs to the configurations and solve the new SMT constraints to find a repair. Clearly, such a SMT-based approach guarantees to provide correct updates but can not scale to large networks, due to the large number of SMT constraints and free variables.

For example, the search space of AED [1] is the power set of delta variables in a syntax tree (Figure 3b), each associated with a node, representing if the corresponding line of configuration is disabled. Additional variables may be required to add more lines or to modify existing values. Thus, the search space is at least 2^{12} for this simple snippet, which will surely be much larger in a hyperscale network.

3 FROM ASR TO ACR

In this section, we discuss two commonly used ASR methods: the *semantics-driven* and the *generative-and-validate* (i.e., *syntax-driven*). We then show the merits of the latter for ACR based on our experiences.

3.1 Approaches of ASR

Semantics-driven approaches formally encode the repair problem as a satisfiability problem and solve the problem with off-the-shelf solvers. The advantage of this approach is that it can guarantee the repair does not have side effects. For example, it can transform the repair problem into a set of SMT constraints and solve it with a Z3 solver [19, 21].

Many existing methods for the synthesis and repair of network configurations can be seen as variants of the above semantics-driven approaches. For example, CPR [13], and AED [1] encode control semantics and operator intents as SMT constraints, and incrementally synthesize configurations by solving a satisfiability solution, whereas CEL [15] localizes the root cause by solving a MaxSAT problem.

Generate-and-validate (Syntax-driven) is another type of approach to ASR. At a high level, it *localizes* a set of suspicious statements, *generates* a set of candidate updates for the suspicious statements, and *validates* them to discard those with side effects. [17, 18, 22] Compared to the semantics-driven method, generate-and-validate is more scalable. This is because the search space is reduced to the set of updates that can be obtained by applying change operators on the suspicious statements.

3.2 Generate-and-Validate for ACR

We believe that the generate-and-validate is a promising way to realize ACR, based on the following three observations:

(1) The search space of ACR is often limited and predictable. In a large enterprise network, the role of each device is relatively clear and fixed, and devices with the same role tend to have similar configurations. This indicates that the same error is likely to be repeated across different places. Thus, it may be possible to use repairs in history to guide

an update for current incidents, which is a similar intuition shared by ASR [18, 22]. *In ByteDance's network, we found only* 9 *types of errors out of over* 100 *incidents, limiting the search space to a small set of historical repair patterns.*

(2) Localization can be achieved based on coverage. Many software fault localizations use test coverage to assess the suspiciousness of a statement [3, 16] based on the intuition that a statement is more likely to be a faulty one if it is covered (executed) by more failed tests, which are the ones that violate the specification. We envision network configurations share a similar intuition and can adapt such a technique to configuration fault localization. To compute coverage, we can use network provenance methods like Y! [26] or coverage computation methods like NetCov [27].

(3) Validation is efficient with incremental network verifiers. Incremental network verification is well-developed [29, 30], which can fast check the correctness of a configuration change for large networks in seconds. This reduces the cost of validating an updateand allows us to try more updates in less time. Note that in ASR, the validation process is often expensive since it needs to recompile and rerun parts or the whole program [23]. Therefore, efficient network verification can ensure the efficiency of the repair system.

4 DESIGN OPTIONS

4.1 Fault Localization

Fault localization narrows down the search space by identifying the statements that seem most suspicious as the root cause. Spectrum-Based Fault Localization (SBFL) [16] is arguably the most popular method.

SBFL scores each statement with suspiciousness, a decimal between 0 and 1, indicating the possibility of being the root cause. This technique assumes that the statement most likely to cause an error is the one most likely to be executed in a failed test case. Therefore, SBFL counts the number of times each statement is executed in all test cases and then applies a formula to calculate the suspiciousness for each statement.

Tarantula [16], one of the most used SBFL techniques, formally defines the suspiciousness of a statement s as:

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

where $failed(s)$ and $passed(s)$ represent the number of failed and passed test cases that are executed by the statement s, respectively; Meanwhile, $totalpassed$ and $totalfailed$ represent the total number of failed and passed test cases.

The effectiveness of SBFL algorithm depends heavily on the quality of the test suite. A test suite with low coverage may miss errors that are not covered by any test case. A classic approach to generate a test suite in ASR is to use symbolic execution to compute a set of inputs that can cover as many code paths as possible [4, 5]. However, we do not have symbolic execution tools for network configurations. Indeed, some verifiers [9, 28, 31] can symbolically execute a control plane model, by making the up/down state of nodes and links symbolic. It is not clear how to extend them to generate tests since this requires making the values in configurations symbolic.

Fortunately, we observe that the specifications to be verified (e.g., reachability, loop-freedom, and blackhole-freedom) in our network already covers most errors of interest, and we can use them to generate test cases. Specifically, each property in the specification includes a header space represented by a 5-tuple. For each property, we sample a packet from its header space as a test, and use a verifier to check whether this property holds. If the property holds, we add the packet to the set of passing tests. Otherwise, we will add it to the set of failing tests.

4.2 Fix Generation

Fix generation defines a set of change operators that limit the search space and uses a heuristics strategy to generate a set of updates by applying change operators on the identified suspicious configurations.

Change operators consist of *atomic operators* and *change templates*. The former modifies one location at a time, while the latter aims to perform multiple changes each time rather than a single one.

For ASR, atomic operators often refer to copying a statement from elsewhere, deleting a statement, or changing the operators or variables in a statement [17]. Network configurations, however, are more concerned with parameters such as IP prefix, AS number, etc., rather than operators or variables. As a result, directly copying existing configuration lines may lead to conflicts (e.g., the same IP addresses are allocated on multiple interfaces), or inconsistency (e.g., the AS numbers of BGP neighbors do not match). To guarantee semantic correctness, we choose to solve for values that can make all previously failed tests pass, based on the SMT constraints collected by symbolic execution.

Such a hybrid approach takes into account the semantics of configurations, but it is still more efficient than a purely semantic-based approach because it uses SMT only when it tries to locally search for a value for a single variable, which may have side effects on other policies globally. Instead, semantic-driven methods need to guarantee the feasible repair has no side effects, which may result in much more variables and constraints.

Templates can be defined manually or extracted automatically. The key is to learn from historical repair experience,

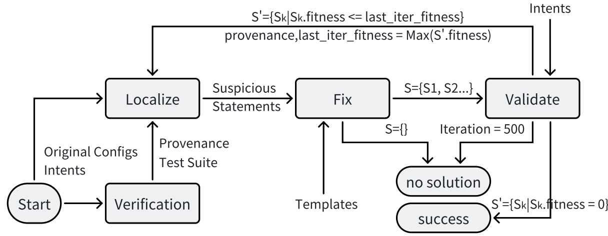


Figure 4: The workflow of ACR.

thereby raising the chances of resolving similar incidents. Consequently, the effectiveness of using templates heavily depends on the coverage of historical incidents. Theoretically, templates can potentially repair any type of error as long as a suitable set of empirical repairs is provided.

Generation strategy consists of *brute-force* and *search-based*. Brute-force systematically applies all change operators to all suspicious statements, defining the search space as the Cartesian product space of suspicious statements and change operators. In contrast, the search-based approach uses a random or heuristic strategy to guide the application of change operators.

One classical search-based approach is the genetic algorithm [17, 22], which takes multiple iterations to evolve candidate updates. In each iteration, the algorithm performs a mutation by applying a change operator to a suspicious statement randomly selected from either the original program or any one of the updated programs from previous iterations. In addition, some methods [17] also randomly select two more statements to perform a single-point crossover to produce two more candidate updates.

Compared to brute-force, a major advantage of the search-based approach is that statements to modify are not limited to the original program, indicating that it can deal with errors requiring to apply change operators multiple times. The downside of the search-based approach is the uncertainty to produce a feasible update. Therefore, ASR often exploits information from history repair [18] to guide the generation of candidate fixes, thus increasing the probability of generating a feasible update.

5 PRELIMINARY DESIGN

The Workflow. We propose localize-fix-validate, a variant of the generate-and-validate approach. The workflow is shown in Figure 4, which generates the feasible update through multiple iterations of evolution. Each iteration contains the three steps of localize, fix, and validate.

Change Operators. For now, we manually define change templates from historical repair experience, focusing on solving the types of misconfigurations shown in Table 1. Specifically, we associate a set of templates with each line of configuration, so that a relevant template can be selected if a line is identified as suspicious. For example, Figure 3c shows the templates (dark rectangle) if a policy-related configuration is identified as suspicious (light rectangle). Each aims to resolve the error type in the Policy category from Table 1 and is implemented by a set of atomic operators (circles). Note that the suspicious configuration line may not always be the line under repair because the "fix place" is determined by the template we choose.

Search Space. The search space of our design is the set of leaf nodes of a search forest (Figure 3c), each representing an atomic change, e.g., symbolize a variable (circles) in the templates (dark rectangles) on a corresponding line of configuration (light rectangles). Although an atomic change involves solving a symbolic variable, the search space is smaller than that of network synthesis because we only need to solve one variable each time. Additionally, the heuristic use of templates avoids exploring irrelevant solutions, further reducing the search space.

Fitness Function. Since a random strategy is not guaranteed to hit an effective change operator, we define a fitness function to measure how good a candidate update is and discard the one that seems *not good*. Specifically, the fitness of an update is defined as the number of failed cases, and candidate updates with high fitness (fitness above the previous iteration) are discarded. The fitness of an iteration is defined as the largest fitness among the preserved updates.

Termination. As shown in Figure 4, the whole repair will terminate in one of the three conditions: (1) a nonempty set of feasible updates is found, i.e., the fitness value is 0; (2) no more candidate updates can be generated, i.e., $S = \emptyset$; and (3) the iteration exceeds the limit, which is currently set to 500.

We show how it works using the example incident:

Step 1: Localize. Currently, we apply Tarantula [16] to localize suspicious configurations. Specifically, we compute the suspiciousness score for every line of each router's configuration, based on the number of lines covered by passing or failed test cases. Here, we only show the results for router A in the right part of Figure 2b. The first three columns indicate the coverage of different test cases, each identified by the subnetwork name, and '•' indicates that this line of configuration is covered/executed. Since the routes for 10.0/16 caused the flapping, it is the only failed case. Thus, we can compute the value of $failed(s)$ and $passed(s)$ for each statement s, and use Equation 1 to calculate the suspiciousness. Here, we can get the highest suspiciousness is 0.67, with

both the value of $failed(s)$ and $passed(s)$ being 1 on line 9. The suspiciousness of each line is shown in the last column.

Step 2: Fix. We select the statements with the highest suspiciousness across all routers, and randomly select a predefined template from the associated set to perform a change. Suppose the templates corresponding to line 9 include peer group and routing policy. As shown in Figure 3c, We can choose the action of '**Symbolize** *Z*' on the right-most template to modify the associate ip prefix-list, i.e., copying line 11 and replacing the '0.0.0.0 0' with a symbolic variable, denoted as var.

We then identify the values of the symbolic variables by performing local symbolism. Specifically, we first perform symbolic execution on the network provenance, that is, we use the precondition of each derivation in the provenance as constraints on the variables, and collect the constraints P that allow passed cases to satisfy intents and constraints F that allow failed cases to violate intents. Then, we obtain the values of the symbolic variables by using an SMT-solver to solve an assignment that satisfies $P \wedge \neg F$. Here, we will have $P : 10.70/16 \in var \wedge 20.0/16 \in var$, and $F : 10.0/16 \in var$, so one possible var can be $\{10.70/16, 20.0/16\}$. Accordingly, the following lines can be inserted before line 11:

ip prefix-list default_all **index** 5 permit 10.70/16
ip prefix-list default_all **index** 5 permit 20.0/16

Step 3: Validate. Currently, we use DNA [29] in the validation to incrementally run test cases and count the number of failed cases. As we argued in §2.3, merely modifying router *A* will create a forwarding loop between *C* and *S*. DNA can capture this loop and identify it as the only failed case. Since this update does not increase the number of failed cases (still 1), this candidate will be kept for the next iteration.

Second iteration. Similar to the localization on router *A*, We compute the suspiciousness for router *C*. The line '**peer** DCNSide **route-policy** Override_All **import**' is one of the most suspicious statements with value of 0.5 (not shown here). Then, we can apply the same change operator in the first iteration, i.e., inserting '**ip prefix-list** default_all **index** 5 permit *20.0/16*' in the corresponding prefix-list block. Due to the change to the override policy, *C* no longer selects *S* as the next hop for 10.0./16, thus the forwarding loop between *C* and *S* can be removed.

Such a preliminary design is scalable since the space for possible updates is relatively small, and the symbolic execution based on provenance avoids path explosion. It is also effective because the values that are solved by SMT constraints reduce the chances to introduce side effects, and multiple rounds of evolution make it possible to handle the misconfigurations at multiple places.

6 FUTURE DIRECTIONS

Hypotheses for ACR. Hypotheses are critical to automatic repair. For example, based on the plastic surgery hypothesis in ASR [6], tools can generate effective repairs by replicating or substituting statement [17]. For ACR, we assume this hypothesis still holds for DCNs, since *devices in DCNs are grouped into several roles, and devices with the same role often have similar configurations*. For other networks like WANs, however, we still need to find other hypotheses, by analyzing the characteristics of their configurations. Only by validating these hypotheses, can we design effective ACR tools for a broader range of networks.

Generating test suite for configurations. In the current design, we leverage the specifications provided by operators to generate test cases. This approach works well for our networks, but may not apply to networks without a specification. For example, operators may not have specifications for the configurations of a legacy network. Therefore, how to automatically generate a test suite with high coverage, so that ACR can apply SBFL to accurately identify the most suspicious statements is an open question to ACR.

Computing suspiciousness scores for configuration. In ASR, researchers have defined various types of suspiciousness scores for SBFL, such as Tarantula, Ochiai [2], Jaccard [8], etc. In this paper, we chose Tarantula for its simplicity and effectiveness. While there may be other suspiciousness metrics that better suit the network settings, finding them is left as future work.

Universal change operators. In this paper, we manually define templates for our networks, based on historical incidents. This may not generalize to the networks of other service providers. It is worthwhile to explore a universal set of syntactic change operators which can deal with a broader range of networks, and cover even new incidents that have never occurred before.

7 CONCLUSION

In this paper, we show some lessons from the real-world incident experience, and propose the need and a preliminary design for automatic configuration repair. This design explores the possibility of a new direction from the field of software repair, called the localize-fix-validate method. We show the feasibility of this method with a typical misconfiguration in a large service provider's network.

Acknowledgement. We thank all the anonymous HotNets reviewers for their valuable comments and suggestions. This work is partially supported by the National Natural Science Foundation of China (No. 62272382). Peng Zhang is the corresponding author of this paper.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*. 49–60.

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D Ernst. 2010. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering* 36, 4 (2010), 474–494.

[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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Aaron Gember-Jacobson, Costin Raiciu, and Laurent Vanbever. 2017. Integrating verification and repair into the control plane. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 129–135.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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].

[25] 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.

[26] 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.

[27] Xieyang Xu, Weixin Deng, Ryan Beckett, Ratul Mahajan, and David Walker. 2023. Test Coverage for Network Configurations. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1717–1732.

[28] 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.

[29] 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.

[30] 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.

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