Diagnosing Distributed Routing Configurations Using Sequential Program Analysis

Rulan Yang[†], Xing Fang[†], Lizhao You[★], Qiao Xiang[†], Hanyang Shao[†],
Gao Han[†], Ziyi Wang[†], Jiwu Shu[†‡], Linghe Kong[◇]
[†]Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University,
[★]School of Informatics, Xiamen University, [‡]Minjiang University, [◇]Shanghai Jiao Tong University

ABSTRACT

In this paper, we show that by capturing the causal relationship among the computation of routers, one can transform the distributed program composed of routing processes into a sequential program, which allows the use of various sequential program analysis theories and tools for diagnosing and repairing routing configuration errors. This insight sheds light on future research on automatic network configuration diagnosis and repair. To demonstrate its feasibility and generality, we give the preliminary design of two methods for routing configuration error diagnosis: (1) data flow analysis using minimal unsatisfiable core and error invariants; and (2) control flow analysis using selective symbolic execution. Using real-world topologies and synthetic configurations, we show that both methods can effectively find errors in routing configurations while incurring reasonable overhead.

CCS CONCEPTS

• **Networks** → **Network reliability**; **Routing protocols**;

KEYWORDS

Network verification, Network diagnosis

ACM Reference Format:
Rulan Yang, Xing Fang, Lizhao You, Qiao Xiang, Hanyang Shao, Gao Han, Ziyi Wang, Jiwu Shu, Linghe Kong. 2023. Diagnosing Distributed Routing Configurations Using Sequential Program Analysis. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, Hong Kong, HK, China, 7 pages. https://doi.org/10.1145/3600061.3600065

Lizhao You and Qiao Xiang are co-corresponding authors.

APNET 2023, June 29–30, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600065

1 INTRODUCTION

Network configuration verification tools [2–5, 11–13, 16, 17, 20, 22, 29–31] analyze the configuration of network devices to decide whether they would compute invariant conforming (*e.g.*, reachability, blackhole freeness, and loop freeness) forwarding rules. Although they are powerful tools for preventing network configuration errors, they answer a binary question: whether the configurations are correct or not. Once they find that the configurations are incorrect, it is still up to the operators to manually find which parts of the configurations are erroneous and fix them, which is both time-consuming and error-prone. For example, given a set of incorrect configurations, Minesweeper [3] returns one counterexample. However, fixing these configurations requires finding all possible counterexamples, which is NP-complete.

Existing studies treat the diagnosis and repair of network configurations as separate issues and propose point solutions with different limitations. Data provenance-based tools [6, 21, 27, 28, 32] analyze the relationship among events in the network to find the root causes (*e.g.*, link failures) of network behaviors (*e.g.*, updates of forwarding rules). However, they are limited to diagnosing only observed events and require reimplementing routing protocols using datalog-based declarative networking languages (*e.g.*, NDLog [21]). CEL [15] extends Minesweeper's SMT-based configuration verification formulation and computes the formula's minimal correction set as the configuration errors. However, it cannot interpret the found errors, which is important for operating real networks. Moreover, it cannot diagnose path-based errors (*e.g.*, waypoint violation) due to the path encoding explosion issue of the SMT-based formulation. Campion [25] focuses on finding the differences between two given configurations of one router, not finding the errors in one set of network configurations. Configuration repair tools (*e.g.*, [1, 9, 14]) compute patches to erroneous configurations without a diagnosis, making it difficult for operators to understand and reason about the patches. In addition, they cannot handle complex settings (*e.g.*, BGP route announcements [14]).

In this paper, we advocate that distributed routing configurations can be diagnosed as sequential programs using causal relationships. As such, the localization and repair of configuration errors are not independent of each other, where the

localized root cause can provide guidance to repair the configurations. Specifically, the routing messages (*e.g.*, BGP route announcement and OSPF link state announcement) from one routing process could trigger actions in another routing process. This causal relationship can be encoded into a directed acyclic graph (DAG), where nodes represent invocations of the computation processes of different routers, instantiated with a specific event and the internal state at the time, and edges represent the (event, action) dependency among the computation processes of different routers.

Given the causal relationship DAG, we make a key insight: *by traversing this causal relationship DAG in topological order, we can transform the execution traces of a distributed system (i.e., the set of individual execution history of each router) into a single execution trace of a sequential program in which each routing process is a function.* This transformation allows us to effectively diagnose and repair network configuration errors because how to analyze a sequential erroneous program is a well-studied problem in the area of program analysis [26].

As a first step to leverage this insight, we design Scalpel, a companion of simulation-based configuration verification tools (*e.g.*, [13, 20, 22, 24]) that automatically diagnose routing configuration errors using two methods:

A data-flow diagnoser using minimal unsatisfiable core and error invariant (§3). The diagnoser first uses satisfiability modulo theories (SMT) to encode the error trace into a conjunction formula of the data plane computation and the desired network behavior (*e.g.*, reachability) that was violated. It then computes a minimal unsatisfiable core (MUC), a conjunctive unsatisfiable SMT formula becoming satisfiable if any of its subsets are removed [19], of the SMT formula. It corresponds to a minimal sequence of router computations during the simulation that leads to the data plane error. Then, it computes the error invariants [10] of this sequence, a set of predicates that explain how these computations lead to the error. Together, the responsible part of router configurations contributing to the minimal computations and the attached error invariants compose a root cause error explanation.

Although the data-flow approach is efficient in slicing the error trace, it can only handle *observable* errors that lead to erroneous paths. It cannot handle *unobservable* errors that lead to the missing required paths. Furthermore, it cannot diagnose latent observable errors, which (1) have not led to erroneous paths yet due to the existence of other observable errors, and (2) will lead to erroneous paths once the latter has been repaired. Diagnosing and repairing only the observable errors. As such, we resort to a second diagnosis method:

A control-flow diagnoser using selective symbolic execution (§4). We extend the simulation process with symbolic routes with symbolic router decisions to diagnose unobservable and latent errors. To cope with the complexity of encoding all possible router decisions, we are motivated

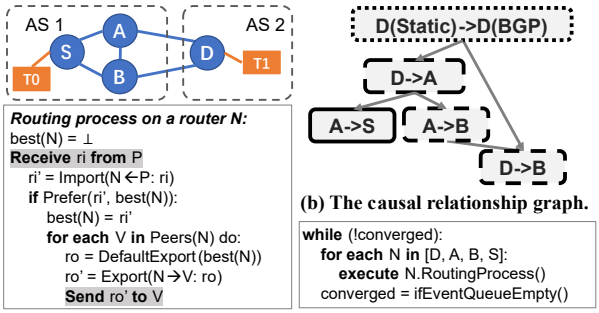


Figure 1: Distributed routing as a sequential program.

by the selective symbolic execution [7] to introduce symbolic routes into the simulation selectively. Specifically, we first compute a requirement-compliant forwarding tree as a reference. We then purposely introduce symbolic routes, which are part of the forwarding tree but are not supposed to be propagated based on the actual configurations, into forked simulation processes to answer "what if" questions, *e.g.*, "what if the router does not filter routes from other neighbors?" or "what if the router chooses route $r1$ over $r2$ as the best route?". As such, the k-failure symbolic simulation of Hoyan [29] is a special case of Scalpel's selective symbolic simulation. Eventually, the simulation would yield at least one ghost execution trace with desired outputs (*i.e.*, requirement-compliant routes) and the corresponding conditions on router decisions. These conditions are a set of router decisions that the correct configurations need to satisfy. As such, the differences between the correct configurations and the actual configurations are diagnosed as errors.

Experiment results (§5). We implement a prototype of Scalpel as a plugin of Batfish [13] and evaluate its performance on real-world network topologies of different scales (*i.e.*, $O(10) \sim O(100)$) with synthetic network configurations. Results show that Scalpel finishes configuration diagnosis within seconds or minutes with a high accuracy.

2 DISTRIBUTED ROUTING AS A SEQUENTIAL PROGRAM

Distributed routing process. We consider the network as a distributed system, where multiple routing processes run on. Routing processes on different devices exchange routing information (*e.g.*, BGP UPDATE message, OSPF link state announcement) using a specific protocol (*e.g.*, BGP, OSPF), and routing processes on the same device may exchange routing information with each other using route redistribution. We abstract the routing process as an event-driven function that is controlled by configurations to compute routes. It receives events such as routing information and local link up/down, and responds to them in a first-in first-serve way to (1) update its internal state (*e.g.*, computing its routing information base and updating its link-state database), and (2) send routing messages to other neighbors if needed. As

such, an event is a *cause* of a router's action (also referred as route computation) to update its internal state and send out derived routing messages, and the action will be the trigger event of other events. In this way, the data plane computation can be defined as the execution of a series of events.

Figure 1(a) shows an example, where there are four routers in the network: S, A, and B belong to the same AS with source prefix $T0$ in S; D belongs to the other AS with destination prefix $T1$. S, A, and B use iBGP to connect to each other; D and A (B) use eBGP to connect to each other. Figure 1(a) also shows the program of a distributed routing process, where a router N receives a route ri from a neighboring router P.

Causal relationship graph. A causal relationship graph (CRG) is a directed acyclic graph (DAG) that indicates the dependency of distributed routing processes. The above routing process relationship can be encoded into a DAG, where nodes represent invocations of the computation processes of different routers, instantiated with a specific event and the internal state at the time, and edges represent the (event, action) dependency among the computation processes of different routers and the state dependency among those of the same router. Specifically, it abstracts each router as an event-driven process and captures the causal relationship of each process during the simulation of the verification tool, by intercepting their input (*i.e.*, inbound route control messages and internal state) and output (*i.e.*, outbound route control messages and new updated internal state). The cause events at routers are route updates from neighbors (*e.g.*, BGP route announcements or link state announcements), and the actions represent how routers handle these announcements: discard, accept, or select (and forward them to neighbors).

CRG is highly dependent on the implementation of a simulator, where different scheduling orders of the routing processes may generate different CRGs. However, given a simulator with a fixed scheduling order, Scalpel can capture a deterministic CRG. Figure 1(b) shows an example CRG with scheduling order (D, A, B, S), where the route announcement A → B happens before the route announcement D → B, and there is no further route announcement from B since the announcement A → B has the higher local preference.

Sequential program. Given a CRG, we can transform the distributed network computation into a linear execution sequence by topologically traversing the CRG. A topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited. Then we can construct a sequential program as shown in Figure 1(c) with scheduling order (D, A, B, S) that generates the same output as the distributed routing protocol. In this way, we turn the diagnosis problem of distributed routing configurations into the diagnosis problem in a sequential program, while the latter is well-studied in the area of program analysis.

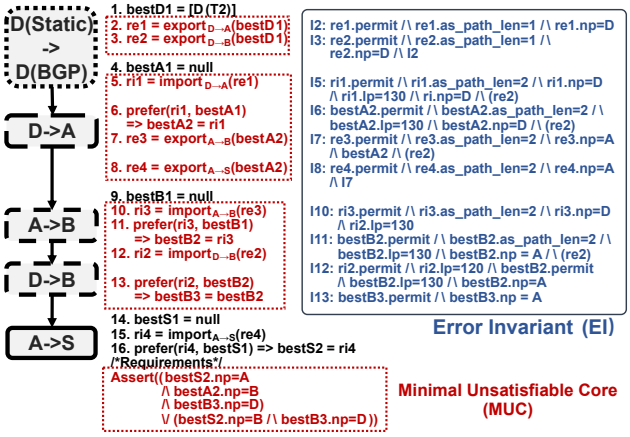


Figure 2: The error trace of the CRG in Figure 1. (We refer to lp as local preference and np as next hop)

3 DATA-FLOW DIAGNOSER

In this section, we show how to diagnose traceable configuration errors using data-flow analysis. In particular, we adopt the minimal unsatisfiable core (MUC) and interpolation approach to diagnose the error trace. MUC is a minimal set of unsatisfiable statements that prove the unsatisfiability of the trace and the interpolation of it is a set of formulas that represent the network states that lead to the error. We reduce the error trace to a minimal set of statements that contribute to the error and compute the interpolation for them to provide the minimal and sound error explanation.

3.1 Error Trace Generation

We first generate an error execution trace from CRG. The linear execution sequence can be encoded as an execution trace with SMT constraints. Informally, a trace is a tuple (π, ϕ) where π is the execution sequence of the network and ϕ is the state formula that describes the desired output state of the trace (*i.e.*, the requirement of the network). The execution of a trace (π, ϕ) is a conjunction of the trace formula $TF(\pi) = T_1 \wedge ... \wedge T_n$ where n is the length of the trace. Let $\pi[i]$ be the i-th statement in the execution sequence, and T_i be the corresponding transition formula. A trace is called an error trace if $TF(\pi) \wedge \phi$ is unsatisfiable.

We use the example in Figure 1(a) to show how to generate an error trace. The requirement is that the traffic from $T0$ to $T1$ should leave AS1 from router B (*i.e.*, S-B-D). However, the actual forwarding path is S-A-D, which violates the requirement. This is because router A sets a higher local preference (*i.e.*, 130) than that set by router B (*i.e.*, 120) for routes from router D. Figure 2 shows the corresponding error trace, including route computation steps in all nodes.

3.2 Configuration Error Localization

We use the classical MACRO algorithm [19] to compute all MUCs, each of which corresponds to one possible error root cause. The algorithm shrinks the unsatisfiable core in order

and iteratively blocks up the computed MUC to enumerate all MUCs efficiently. The red rectangular in Figure 2 shows an example MUC, which slices the error trace.

Another issue is how to map the MUC to the execution sequence. The execution sequence $\pi[1, ..., n]$ is translated into the trace formula $TF(\pi) = T_1 \wedge ... \wedge T_n$. Let T_i consist of a set of clauses $c_1 \wedge ... \wedge c_m$. We add a new boolean constraint p_i to track these clauses so that we can map them to the initial statement $\pi[i]$, *i.e.*, replacing c_j with $p_i \wedge (p_i \implies c_j)$

3.3 Configuration Error Explanation

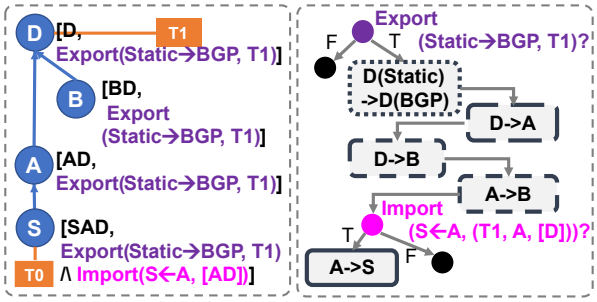
Given a pair of formulas A and B where $\neg(A \wedge B)$ holds, an interpolant [8] of A and B is a formula I over the common symbols of A and B such that $A \implies I$ and $B \implies \neg I$. It indicates the inconsistency between the pair of formulas. Given an indexed sequence $\Phi = [A_i]_{i=1}^{n}$ such that $A_1 \wedge ... \wedge A_n = false$ (*e.g.*, the execution sequence), the interpolants for Φ is a sequence $I_0, I_1, ..., I_n$ which labels each A_i with inconsistent variables that explain the unsatisfiablility of Φ.

We generate the error explanation for the error trace by interpolating MUC. Specifically, we compute interpolants along the MUC slice based on the hybrid algorithm [23] and preserve the dependency of specific variables (*i.e.*, configuration-related variables), so that we can observe which configuration variables contribute to the error.

Then we localize configuration errors from the error explanation. Each interpolant in the interpolation sequence of MUC is a formula that captures the intermediate state that produces the error. Since the execution trace is a serial of route computations, interpolants can explain why an erroneous route is generated and the contributing configuration variables. The right side of Figure 2 shows the error explanation of the violation of the way-pointing requirement for S (traffic from $T0$ to $T1$). First, D generates a target route to $T1$ and propagates it to peer A and B (the first red dash line box); node A then receives the route from D and selects it as the best route since A has only one route, and propagates it to peer B and S; third, B receives routes from A and D respectively and selects the route from A as the best route because of higher local preference (*i.e.*, $bestB2.lp = 130$ and $ri2.lp = 120$ as the explanation). Finally, the requirement (*i.e.*, assertion) is violated, since the requirement specifies the next hop of B must be D.

4 CONTROL-FLOW DIAGNOSER

We leverage control flow analysis to diagnose unobservable errors. This is because the route computation data flow of such errors is usually *unobservable* (*e.g.*, the route propagation will terminate prematurely once the target route is filtered by the origin redistribution process). We view the data plane computation as a program that consists of route propagation and computation and is controlled by network configurations. Symbols of the program are router decisions



(a) The propagation of symbolic routes. (b) The execution path.

Figure 3: Symbolic execution of the example network.

such as the delivery of routes and the transformation of routes. Configuration errors that contribute to the deviations of router decisions will not surprisingly result in incorrect route computation outputs (*i.e.*, incorrect forwarding behaviors of routers). So our goal is to find *a minimal decision deviation set* by symbolically executing the data plane computation to localize configuration errors.

We use the same network topology in Figure 1(a) to present details. Assume there are multiple errors: (1) D has not configured the network command to redistribute the target route to its BGP process, (2) S filters route to $T1$ from both peers A and B. The requirement is: $T0$ and B should reach $T1$.

4.1 Selective Execution Conditions

We efficiently execute the symbolic data plane computation using selective execution conditions to avoid exploring huge execution path space. We first introduce how to generate such conditions using the forwarding tree representation. Each forwarding tree is a directed graph toward a prefix that specifies the forwarding paths of all nodes. The reversal of it indicates the route propagation and selection conditions that nodes need to comply with. We construct such a tree based on previous causal relationships to avoid incurring unnecessary interference. Specifically, we first keep existing policy-compliant paths in the tree. Second, we complete paths for remaining source nodes that do not satisfy requirements. Figure 3(a) shows the forwarding tree of the network: node S and node B forward traffic to $T1$ along with path S-A-D and path B-D respectively.

Priority condition for route selection. To ensure that each node i eventually follows the path in the forwarding tree to propagate traffic, we use prefix, next-hop, and as-path attributes to identify the target route for node i. The priority condition is that i always chooses the target route as the best route (*i.e.*, has the highest priority) for the destination prefix.

Export condition for route delivery. Each node i is obligated to propagate the route to its in-neighbors so that its in-neighbors can receive the target route. Additionally, if node i adjacent to its out-neighbor n_{out} and in-neighbor

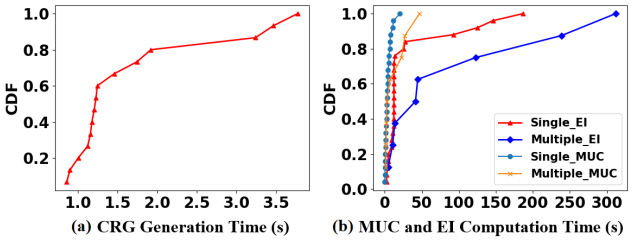


Figure 4: Running time on diagnosing observable errors.

n_{in} uses different protocols, node i needs to redistribute the target route to the routing processes of nodes i and n_{in}.

Import condition for route delivery. The import condition follows the same intuition as the export condition: ensuring the target route is delivered to the corresponding node in the forwarding tree. Specifically, node i has to receive the target route from its out-neighbor rather than filter it. If a node has no out-neighbor, the condition for it is to filter all routes from adjacent peers.

Adjacency condition between routers. The adjacency condition is a pre-condition for route deliveries between routers. We define the adjacency condition for nodes that have out-neighbors in the forwarding tree as the node needs to establish the peer session with its out-neighbor.

4.2 Symbolic Data Plane Computation

We define four types of conditions (*i.e.*, router decisions) as symbols when executing them. These conditions are the abstraction of router decisions that are sufficient to result in one of the error-free data planes towards the prefix. To improve efficiency, we do not keep all execution paths triggered by the above conditions. We tend to always guard the newest execution path that satisfies all conditions in this way: once a condition is triggered and the actual router decision deviates from it, we enforce the condition satisfiable and record it as the deviation condition of this path, and then continue to execute the computation. Figure 3(b) shows the correct execution path that results in the same data plane as the forwarding tree. We only symbolically execute two symbols during the computation: export and import decisions on nodes D and B, respectively (the purple and pink branches). We forcibly assign them the true values so that the divergences are the deviations of router decisions.

4.3 Configuration Error Localization

After symbolic data plane computation converges, each route carries these symbols to represent its existing conditions (concrete conditions), as shown in Figure 3(a). The symbolic conditions are the minimal deviation set of router decisions that indicates the causes of the error. We map their keywords to defined types and localize configuration errors.

5 PERFORMANCE EVALUATION

We implement a prototype of Scalpel as a plugin of Batfish [13] in 6K LoC in Java and Python. For observable errors, we use Z3 as the SMT solver to encode the error trace and compute MUCs to localize errors. For unobservable errors, we integrate the selective symbolic execution into the data plane simulation to localize errors. All the experiments are performed on a Linux server with two Intel Xeon Silver 4210R 2.40GHz CPUs and 128GB of DDR4 DRAM.

Network	#Reqs	#MUC+EI	#Clause	Len
Arnes	2/4	2/10	287/450	11/23
Bics	1/4	4/1	301/666	11/4
Canerie	1	1	236	21
Renater2008	1/4	2/4	555/522	12/4
Columbus	1/4	2/4	602/368	62/7
Colt	1	1	1052	18
Cogentco	1	10	1217	7
UsCarrier	1	3	731	19

Table 1: Diagnosis results on observable errors.

5.1 Methodology

To demonstrate the soundness and correctness of the diagnosis results Scalpel provided, we run experiments on real topologies from Topology Zoo [18].

Datasets. We use 10 synthesized networks ranging from O(10) to O(100) nodes and as datasets. We synthesize BGP configurations ranging from 3,000 to around 20,000 lines written in Cisco's IOS language using NetComplete [9].

Requirements. Each requirement is a packet forwarding path towards a pair of (source, destination) nodes and the reachability of all nodes to one destination node for observable and unobservable errors, respectively.

Injecting configuration errors. In order to demonstrate that Scalpel can find and correct configuration errors, we manually introduce errors to node configurations and record the configuration lines we changed as the errors in configurations. We communicate with network operators in a famous network company and conclude three common errors in the network with BGP protocol: wrong local preference value, route denial when propagation, and route denial from origin. To inject configuration errors, we randomly add, delete or modify some lines in node configurations (e.g., set local preference to another value, add route maps to deny imported routes, or just delete configuration segments in origin configurations). Furthermore, we modify the original path requirement: we select another packet forwarding path from source to destination as the requirement, and generate new configurations as the erroneous configuration corresponding to previous requirements.

5.2 Experiment Results

Overhead of capturing casual relationships. Our Scalpel prototype modifies the source code of Batfish [13] to capture the causal relationship during simulation. The causal relationship is recorded as logs and stored in files. Then we use the logs to construct CRG and encode the error trace.

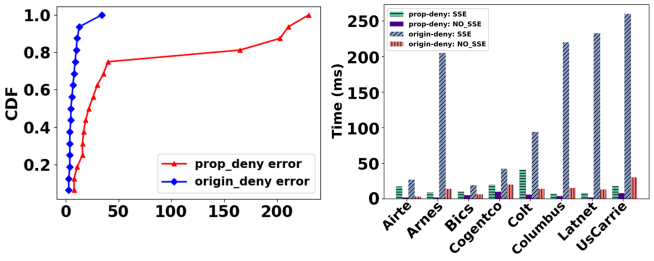


Figure 5: Running time on diagnosing unobservable errors.

We measure the time of recording causal relationships and generating CRGs. Figure 4(a) shows that 80% of networks' CRGs can be generated in 2s. We find that the overhead increases linearly with the increasing size of the topology as larger size topology always has a longer iteration process. To summarize, capturing causal relationships takes little time in Scalpel's whole process.

Efficiency. We capture Scalpel's diagnosis results on error traces on different topologies with different numbers of requirements in Table 1. We substitute clauses with shorter MUCs, shrinking the scope for repair. There always exists more than one MUC and EI pair in a single error trace because there could be more than one explanation for the error.

Figure 4(b) gives the time of computing the single and all MUC/EI(s). Scalpel computes single MUS $\leq$ 10s in 90% cases. We compute up to ten MUCs for each test, and all tests are finished in $\leq$ 50*s*. Computing EI is more time-consuming. It takes $\leq$ 30*s* ($\leq$ 50*s*) to compute a single (all) EI in 80% (60%) of cases. In the worst case, it takes 340*s* to compute all EIs.

We plot the execution time for two kinds of unobservable errors in Figure 5. Diagnosing origin-deny errors in each network finishes in less than 50ms. For most networks containing route propagation errors, SSE takes less than 50ms for most cases and up to 230ms. Diagnosing origin-deny errors tends to take less time than other errors. We guess it is because SSE could find the "origin" error at the very beginning of the simulation.

Accuracy. We then leverage the accuracy of the diagnosis results by repairing the network with the diagnosis results that Scalpel provides. For observable errors, Scalpel outputs pairs of MUC and EI, and we have checked that Scalpel locates all local preference errors we introduce to the configurations.

Table 2 gives the diagnosis results on unobservable errors. We form the result in bituple corresponding to (prop-deny error, origin-deny error). RE, FP and FN represent the reported errors, false positive errors, and false negative errors, respectively. Scalpel can diagnose different types of errors, especially for the origin-deny type where no miss detection occurs. For the false positive errors in the diagnosis results, although it seems inconsistent with the injected errors, it may also be a valid but different diagnosis method for the same erroneous data plane. In fact, this kind of result may not be counted as an error, if the network can be repaired using the diagnosis results.

Network	#Errors	#RE	#FP	#FN
Airte	(1,2)	(1,2)	(0,0)	(0,0)
Arnes	(1,2)	(1,4)	(0,2)	(0,0)
Bics	(2,2)	(3,3)	(1,1)	(0,0)
Cogentco	(2,2)	(10,10)	(8,8)	(0,0)
Columbus	(2,2)	(1,2)	(0,0)	(1,0)
Latnet	(2,2)	(2,2)	(0,0)	(0,0)
UsCarrier	(1,2)	(1,2)	(0,0)	(0,0)

Table 2: Diagnosis results on unobservable errors.

6 DISCUSSIONS

Supporting link-state routing protocol. So far, we have discussed the configuration error diagnosis using the BGP protocol (*i.e.*, path-vector protocol). Another popular routing protocol is the link-state protocol (*e.g.*, OSPF and IS-IS), which has been implemented in several simulation-based verifiers. Scalpel can also diagnose these protocols as sequential programs if the CRG can be captured.

Supporting networks running multiple protocols. Multiple routing protocols often co-exist in the same network. For example, the OSPF protocol serves as the underlay to connect peering IP addresses, and the BGP protocol serves as the overlay to exchange routing information. Our approach can be extended to support such settings. The data-flow diagnoser can analyze the error trace generated by multiple routing protocols. The control-flow diagnoser can be extended to include more symbolic conditions.

How to choose between data-flow diagnosis and control-flow diagnosis? Similar to the case in program analysis where data-flow and control-flow analysis each have their pros and cons, so are data-flow and control-flow diagnosis. In particular, control-flow diagnosis can diagnose unobservable errors. Yet its scalability in large networks with complex configurations is still an open question we are investigating.

Repair of erroneous configurations. This workshop paper focuses on diagnosing erroneous configurations. In our ongoing study on configuration repair, we continue leveraging the key insight of transforming distributed routing computing into a sequential program and apply program repair techniques (*e.g.*, mutation-based repair and constraint-based repair) or domain-specific repair techniques (*e.g.*, incremental synthesis) to fix the diagnosed root cause.

Acknowledgments. The authors are extremely grateful for the anonymous APNet'23 reviewers for their wonderful feedback. We also thank Hongyu Du, Franck Le, Ruzica Piskac and Thomas Wies for their help during the preparation of this paper. This work is supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award #62172345, MOE of China Award #2021FNA02008, Open Research Projects of Zhejiang Lab #2022QA0AB05, and NSF-Fujian-China 2021J05003 and 2022J01004.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Aed: Incrementally Synthesizing Policy-compliant and Manageable Configurations. In *CoNEXT'20*. ACM, 482–495.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *NSDI'20*. USENIX, 201–219.

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM'17*. ACM, 155–168.

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

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. In *POPL'19*. ACM, 1–27.

[6] Ang Chen, Chen Chen, Lay Kuan Loh, Yang Wu, Andreas Haeberlen, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2018. Data Center Diagnostics with Network Provenance. *IEEE Data Engineering Bulletin* 41, 1, 74–85.

[7] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective Symbolic Execution. In *HotDep'09*. IEEE.

[8] William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic* 22, 3, 269–285.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *NSDI'18*. USENIX, 579–594.

[10] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error Invariants. In *FM'12*. FME, 187–201.

[11] 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 *OSDI'16*. USENIX, 217–232.

[12] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *NSDI'05*. USENIX, 43–56.

[13] 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 *NSDI'15*. USENIX, 469–483.

[14] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically Repairing Network Control Planes Using An Abstract Representation. In *SOSP'17*. ACM, 359–373.

[15] Aaron Gember-Jacobson, Ruchit Shrestha, and Xiaolin Sun. 2022. Localizing Router Configuration Errors Using Minimal Correction Sets. *arXiv:2204.10785. Retrieved from https://arxiv.org/abs/2204.10785*.

[16] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM'16*. ACM, 300–313.

[17] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient Verification of Network Fault Tolerance Via Counterexample-guided Refinement. In *CAV'19*. Springer, 305–323.

[18] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9, 1765–1775.

[19] Mark H Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. 2016. Fast, Flexible MUS Enumeration. *Constraints* 21, 2, 223–250.

[20] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully Emulating Large Production Networks. In *SOSP'17*. ACM, 599–613.

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

[22] Nuno P Lopes and Andrey Rybalchenko. 2019. Fast Bgp Simulation of Large Datacenters. In *VMCAI'19*. Springer, 386–408.

[23] Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. 2014. What Gives? A Hybrid Algorithm for Error Trace Explanation. In *VSTTE'14*. Springer, 270–286.

[24] Bruno Quoitin and Steve Uhlig. 2005. Modeling the Routing of an Autonomous System with C-BGP. *IEEE network* 19, 6, 12–19.

[25] 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 *SIGCOMM'21*. ACM, 748–761.

[26] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8, 707–740.

[27] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *NSDI'17*. USENIX, 719–733.

[28] 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, 383–394.

[29] 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 *SIGCOMM'20*. ACM, 599–614.

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

[31] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic Router Execution. In *SIGCOMM'22*. ACM, 336–349.

[32] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient Querying and Maintenance of Network Provenance at Internet-scale. In *SIGMOD'10*. ACM, 615–626.