Localizing Router Configuration Errors Using Unsatisfiable Cores

Ruchit Shrestha,* Xiaolin Sun,* Aaron Gember-Jacobson
Colgate University
{rshrestha, xsun, agemberjacobson}@colgate.edu

Abstract

Router configuration errors are unfortunately common and difficult to localize using current network verification tools. Consequently, repairing a network—either manually or automatically—can be time consuming. We present a technique that uses unsatisfiable cores from SMT-based network models to accurately identify which parts of a network's configurations are likely the cause of requirement violations.

1 Motivation

Many networks rely on distributed routing protocols to decide how packets are forwarded through the network. The protocols' decisions are influenced by relationships, preferences, and filters expressed in router configurations. Prior work has shown these configurations are often complex, due in part to the low-level of abstraction exposed by router APIs and the breadth of requirements—e.g., reachability, waypointing, and path preferences—networks must satisfy [2].

The complexity of router configurations has two negative implications. First, it is difficult for network operators to correctly update configurations to satisfy new requirements or improve the implementation of existing requirements; consequently, configuration errors are common. Second, locating errors within configurations is difficult.

To help address these issues, researchers have developed several tools for verifying [1, 4, 6] and repairing [3, 5] configurations. These tools construct a model of the network's semantics—e.g., a system of Satisfiability Modulo Theories (SMT) constraints—and use the model to check whether the network satisfies certain requirements. If a requirement is violated, a verifier outputs a counterexample that demonstrates the violation, whereas a repair tool outputs a configuration patch that corrects the violation.

Limitations of current tools. While these tools have shown significant promise, we argue that they suffer from a fundamental flaw: *insufficient fault localization*. In software engineering, fault localization is the process of identifying which lines of a program likely cause certain test cases to fail [8]. This information is useful, because it can significantly reduce the space of possible code fixes and, correspondingly, the time required to (automatically) repair the program.

State-of-the-art network verification/repair tools fail to provide/leverage such information, thus making it more difficult for humans and machines to repair configurations. In particular, state-of-the-art verifiers [1, 4, 6] provide a single forwarding path and environment (e.g., set of failed links) that demonstrate a violation, but they do not indicate: (1) which portions of the configurations influenced the computation of the forwarding path; (2) whether routers on the path, off the path, or both are at fault; (3) whether violations of the same requirement may manifest in different ways under different failure scenarios; and (4) whether violations of different requirements are related—e.g., caused by the same error. State-of-the-art repair tools [3, 5]—which produce patches rather than counterexamples—consider the entire space of possible repairs and rely on general SMT heuristics to narrow the search space. Consequently, these tools are unable to scale to networks with many routers or requirements.

Objective and challenges. Our goal is to *design a technique for accurately localizing errors in network configurations*, thus paving the way for faster network repair. This requires overcoming unique challenges that are not readily addressed by existing software fault localization techniques [8]. First, network configurations are the *input* to a router's control software. The software itself is typically closed source and out of a network operator's control. Hence, unlike software fault localization, we seek to localize errors in a program's input. Second, routing protocols and configurations are inherently distributed. We must consider multiple programs, each with its own input, running and passing messages in parallel. Thus, network fault localization is more complex than analyzing a single- or even multi-threaded program.

2 Approach

Our key insight is to localize configuration faults using *unsatisfiable cores generated from SMT-based models of network configurations' semantics*.

SMT-based network models. These models [1, 3] abstract away the details of router control software—which is not the target of our fault localization—and encode a network's distributed decision processes as a single system of logical formulas. The models include symbolic variables representing route advertisements and forwarding rules and constraints on these variables representing route filters, preferences, and selection procedures. The latter are closely tied to configurations' syntax, which makes SMT-based models better suited for fault localization than graph-based models [4, 5, 6]. By introducing additional constraints that encode a network requirement and using an SMT solver to check whether the problem is satisfiable, we can determine whether the requirement is violated. It is important to note that Minesweeper [1]

*Undergraduate student author

actually encodes the *negation* of network requirements, such that the problem is satisfiable iff a requirement is violated; a satisfying solution thus represents a counterexample.

If the SMT problem is unsatisfiable, the solver produces an *unsatisfiable core*: i.e., a set of incompatible constraints. Due to Minesweeper's use of negated requirements, the problem is unsatisfiable iff a requirement is always fulfilled. The constraints in an unsatisfiable core thus represent route selection procedures and configuration segments that together lead to correct network behavior.

Generating useful unsatisfiable cores. Since unsatisfiable cores are associated with correct network behavior, it may seem as though they are unsuitable for locating the cause(s) of requirement violations. However, by obtaining unsatisfiable cores for violated requirements and considering unsatisfiable cores' inverse, we can successfully localize faults.

To obtain unsatisfiable cores for violated requirements, we force the SMT solver to "ignore" all counterexamples (i.e., satisfying solutions). In particular, we iteratively solve and add a constraint—specifically, the negation of the solution—to the problem, until no more counterexamples exist and the problem is unsatisfiable. However, since the problem was satisfiable prior to adding these constraints, the solver is likely to produce an unsatisfiable core that includes one of the negated solutions, which does not convey anything about the correctness of configurations. Consequently, we restrict the constraints the solver may include in an unsatisfiable core to those contained in the original problem, thus forcing the solver to expose a "useful" unsatisfiable core.

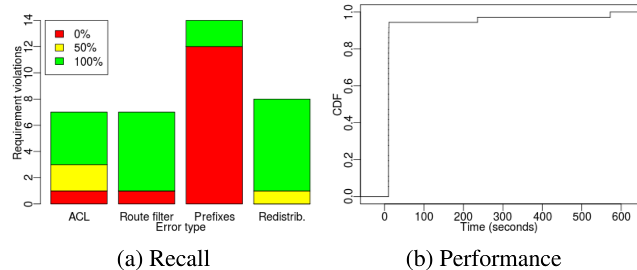
Given an unsatisfiable core for a violated requirement, we identify potentially faulty constraints by computing the set difference between the full set of constraints and the unsatisfiable core. We also ignore all constraints that encode route selection procedures, which are determined by protocol standards, not configurations.

Multiple, minimal unsatisfiable cores. Using the basic approach described above can result in fault localization that both under- and over-estimates which configuration segments likely contain errors. Under-estimation can arise when the SMT solver produces a non-minimal unsatisfiable core, which includes constraints that do not contribute to the problem's unsatisfiability or, equivalently, correct network behavior. Over-estimation can arise due to the SMT solver producing only one, out of many, unsatisfiable cores. Hence, we may overlook constraints that contribute correct network behavior, and assume more configuration segments are faulty.

To address these issues, we compute all minimal unsatisfiable cores using the MARCO algorithm [7]. We then compute the set difference between the full set of constraints and the union of all minimal unsatisfiable cores.

3 Preliminary Results & Future Work

We have implemented a prototype of our technique in Minesweeper [1]. We evaluate its accuracy using six synthetic networks. Each network contains 3 to 20 routers and employs some combination of BGP, OSPF, static routes, and route redistribution to enable reachability between several pairs of subnets connected to different routers. We synthetically introduce errors into these configurations by adding access control lists (ACLs), adding route filters, removing advertised prefixes, and disabling route redistribution.



(a) Recall (b) Performance

Figure 1: Evaluation results

We measure our techniques' recall, precision, and performance. Figure 1a shows for each type of error the fraction of reachability requirement violations for which our technique identified none, half, or all of the SMT constraints that are derived directly from faulty configuration stanzas; our technique effectively detects three of the four types of errors. For all requirement violations, our technique's precision is 100%. Figure 1b shows that our technique takes about 10 seconds to localize faults for 90% of the violations; fault localization takes the longest in our 20-router fat-tree topology.

In the future, we plan to evaluate our technique on real network configurations and additional types of errors. In addition, we plan to explore how to localize faults at a sub-constraint granularity to narrow our localization to individual lines of configuration, rather than configuration stanzas.

References

[1] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *SIGCOMM*, 2017.

[2] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[3] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *NSDI*, 2018.

[4] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.

[5] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. Liu. Automatically repairing network control planes using an abstract representation. In *SOSP*, 2017.

[6] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.

[7] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2), 2016.

[8] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8), 2016.