Indirect Network Troubleshooting with The Chase

Mubashir Anwar[†], Fangping Lan[*], Anduo Wang[*], Matthew Caesar[†]
[*] Temple University, [†] University of Illinois at Urbana-Champaign

ABSTRACT

The future of static verification in networking may be obscured by two clouds: the complexity of distributed systems with highly concurrent events, and the decision-making on infrastructures growing without a premeditated plan. This poster discusses a possible solution to these issues, in which the huge space of analyzing distributed systems and the macro-questions of system evolution are addressed by a common structure, a logical implication problem which we call *indirect troubleshooting*. The usefulness and feasibility of indirect troubleshooting is illustrated by a preliminary realization with *the chase*, a remarkable process for mechanically deciding implications.

1 INTRODUCTION

Network diagnosis and troubleshooting tools used to be the most advanced aid in debugging networks just as testing once was in software, until the rise of software-defined networking (SDN), when static verification took off. What static verification lacks in breadth at first — e.g., early tools unanimously focusing on reachability — it makes up in completeness. By comprehensively exercising the network on an abstract model (e.g., symbolically testing all forwarding paths in a proper data structure), static verification establishes the "existence" of a target invariant in the network, even when the violation at runtime is latent or rare. As network models become more expressive (admitting richer semantics), the exploration algorithms faster (exploiting symmetry etc.), and new services (e.g., coverage metrics) emerge, static verification seems to be on a parallel route to dominate network analysis as program verification subsumed testing. This bright future, however, may be obscured by two clouds.

Cloud I. Networks are by nature distributed systems with highly concurrent events. This key complexity is to some extent defined away by SDN, and program verification[1] alike. Take the popular verification primitive (a typical invariant) "does the program (network snapshot) as exercised by this test input (data packet) exhibit that characteristic?" as an example: when adopted in networking, the primitive translates to "instantaneous verification" of the network at a particular time instance while at a definite state (e.g., forwarding), a task that treats distribution (e.g., route announcements from a neighbor) and concurrency (e.g., link failure) as external environments, which are often abstracted away in the network model and left out during model exploration. While a single run of the primitive completes the verification task on a centralized sequential software, fully checking every relevant network "incarnation" may involve (potentially exponentially) many runs. Not surprisingly, several recent developments have started treating such pain points.

Cloud II. Networks are an ever-growing live infrastructure. Unlike circuits, chips, or software development, where static verification is applied to a clean-slate with well-defined goals in a pre-fixed life cycle, network infrastructures continue to operate and evolve, under constant failures, over a span of decades, without a well-laid plan. If static verification tools were to facilitate decision making for such infrastructures, the network operators, oftentimes non-programmers, would have to retrofit macro-questions like "Is it safe (or impossible) to add a new invariant (policy) given existing ones? Is this failure a sure symptom of this concern? Does an error detected here also signal a potential failure in some remote end of the network?" into lower-level verification primitives for detailed packets, paths etc. While similar inference problem based on some notion of causality was addressed in several early troubleshooting systems [2], a more rigorous formulation with static verification, however, is still missing.

In light of these difficulties, we propose a new primitive we call *indirect troubleshooting* based on the elegant data dependency tool called the chase [1]. Indirect troubleshooting decides if known facts about the network $\sum$ imply other unknown property γ, denoted $\sum \vdash \gamma$. $\sum = \{\sigma_1, \sigma_2, \cdots\}$ is a set of "premises" we begin with, obtained from existing tools that check the network states *directly*. These premises may describe an easily verifiable policy about a network partition, a shared belief about a subset of failures, or a detected error signaling a larger fault, to name a few examples. γ is the "conclusion" we seek to *indirectly* derive. It expresses a more comprehensive property about the network as a whole that existing tools cannot assess. This can include an evaluation of policy changes to aid network evolution, an assertion of the impact of faults to guide network diagnosis, etc. For indirect troubleshooting, the conclusion γ can be mechanically modified to reflect another dependency (premise), σ, using *the chase*, i.e. $\texttt{chase}(\gamma, \sigma)$, which is to check if σ implies γ. This poster illustrates how the ambitious goal of indirect troubleshooting can feasibly be realized by adapting the chase.

2 MOTIVATING EXAMPLE

Consider the problem of distributed management of network policies (invariants) as shown in Figure 2 (a) with three invariants: one security policy that blocks packets from `10.2` to `10.4`, implemented by filters at `1` and `2`; and two rewriting policies that change the source header at `1` and destination header at `2` as shown in the figure. Notably, `1` and `2` can be viewed as two separate management partitions, within which the packet filter is properly enforced alongside a legitimate rewrite. But these two *correct* partitions, when combined, does not securely block traffic. Figure 2 (b) shows the trace of a packet originated at `10.2` with source `10.2` and destination `10.3` that successfully bypasses the filters and eventually reaches

[1] A rich literature on verifying distributed systems does exist. Here we focus on program verification designed for sequential programs, which is the main inspiration for network verification in the SDN era.

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

ACM ISBN 979-8-4007-0782-7/23/06.
https://doi.org/10.1145/3600061.3603137

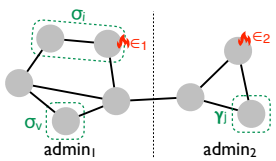

Figure 1: Example: verified invariants, detected errors.

scenario	decision problem	test
evolution assessment	is new policy σ_v redundant (impossible)?	chase($\sigma_v, \sum$) = $\top$ ($\bot$)?
symptom diagnosis	admin1: is remote ϵ_2 bound to occur if ϵ_1 is detected?	chase($\epsilon_2, \{\epsilon_1, \sum\}$) = $\top$?
	is detected failure ϵ_1 a sure symptom of σ_i?	chase($\sigma_i, \{\epsilon_1, \sum\}$) = $\bot$?
impact analysis	is σ_i unaffected when a new error ϵ_1 is detected?	chase($\sigma_i, \{\epsilon_1, \sum\}$) = σ_i?

Table 1: Example questions (decision problems) that the chase can answer to guide the admin of domain 1 in her interaction with existing system troubleshooting and static verification tools. Here, $\sum = \{\sigma_1, \cdots, \sigma_i, \cdots\} \cup \{\cdots \gamma \cdots\}$ is the set of all invariants (verified with existing tools).

10.4, revealing a hard-to-predict interaction between the distributed invariants.

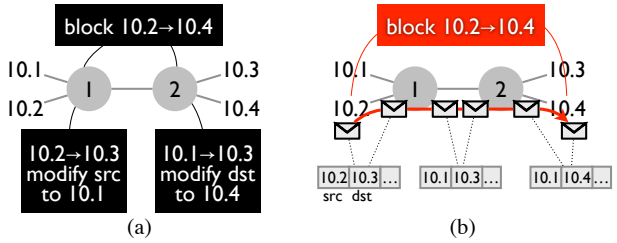

Figure 2: Correct configuration for each network partition (filtering properly enforced for node 1 and 2 separately) does not ensure intended packet filtering on the global network.

From a management and troubleshooting perspective, distributing the enforcement of invariants can help scale the management of complex networks. However, this practice also introduces a new challenge: as a collection of invariants correctly implemented in each partition separately, it becomes essential to determine what properties still hold on the network as a whole. This translates to the implication problem of indirect troubleshooting, which can be solved using the chase.

3 TOWARDS A FLEXIBLE FRAMEWORK

With indirect troubleshooting, we envision a framework that can play an important role in managing network failures and evolutions. Figure 1 illustrates this potential: in the two administrative domains, each is maintaining a set of invariants (policies) — σ's in domain 1 and γ's in domain 2 — with the help of some verifiers, and simultaneously checking for ad-hoc errors (e.g. $\epsilon_{1,2}$). In particular, the admin of domain 1 with restricted visibility can use the chase to answer questions shown in Table 1. It's important to emphasize that our framework is highly adaptable and can work with a wide range of existing tools. For instance, when it comes to identifying individual failures on a real network through system troubleshooting, our framework can provide valuable insights by answering critical questions such as, "Based on the detected failures, are there any other potential failures that could be more severe? And are there any important network properties that remain unaffected?" Similarly, for analyzing complex, large-scale networks using static reasoning, our framework can effectively integrate the results obtained from traditional tools used for smaller network partitions or lower levels of concurrency. This way, we can obtain a more comprehensive understanding of the network and its behavior, leading to better-informed decisions and more efficient troubleshooting.

4 PRELIMINARY RESULTS

We use early stage experiments to quantify the overhead and advantage of indirect troubleshooting with distributed invariants on an example similar to Figure 2, compared to a prior static verification tool, Batfish [3]. We use AS 7018 from the Rocketfuel dataset as our topology, select nodes at random as sources and destinations, and embed IP rewriting and firewall policies along the shortest paths between them. We then verify the packet filtering policies. Figure 3 depicts the running time of verifying distributed invariants on varying number of hosts where the attacks on the distributed firewalls may be launched. Batfish has to check flows between all pairs of source and destination nodes under user's manual guidance. The chase, on the other hand, takes the distributed invariants as dependencies, and automatically detects the security hole as captured in the conclusion dependency (i.e. the filtering policy). Our naive implementation of the chase consistently outperforms the highly-optimized Batfish on all inputs. As the number of hosts increase, the number of unique flows and consequently the time taken by Batfish to exhaustively search the trace of the attack increases. While the chase exhibits a similar slow-down, we stress that the chase looks for the security hole by modifying the conclusion in the network automatically instead of considering each flow one-by-one.

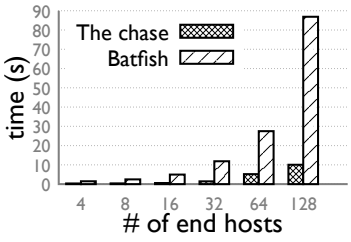

Figure 3: Comparing the chase and Batfish for checking distributed invariants

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation Awards CNS-1909450, CNS-2145242.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Mike Y. Chen, Anthony Accardi, and Dave Patterson. 2004. Path-Based Failure and Evolution Management. In *First Symposium on Networked Systems Design and Implementation (NSDI 04)*. USENIX Association, San Francisco, CA.

[3] 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 *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 469–483.