The Case for Validating Inputs in Software-Defined WANs

Alexander Krentsel[†*], Rishabh Iyer[†], Isaac Keslassy[‡], Sylvia Ratnasamy[†*], Anees Shaikh[*], Rob Shakir[*]
[†]UC Berkeley [‡]Technion [*]Google

Abstract

We highlight a problem that the networking community has largely overlooked: ensuring that the inputs to network controllers in Software-Defined Network (SDN) WANs correctly reflect the state of the network. We show that "incorrect" inputs are a common cause of major outages in production and propose new directions to address these.

CCS Concepts

• **Networks** → *Control path algorithms*; *Network design principles*; **Network reliability**; Network manageability.

Keywords

Software-defined Networking, Traffic Engineering, Input Validation, Verification

ACM Reference Format:
Alexander Krentsel, Rishabh Iyer, Isaac Keslassy, Sylvia Ratnasamy, Anees Shaikh, Rob Shakir. 2024. The Case for Validating Inputs in Software-Defined WANs. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696874

1 Introduction

Availability is a network operator's highest priority and yet, despite considerable effort and investment, state-of-the-art networks continue to exhibit regular outages. For example, recent papers report that the frequency of major outages in large cloud provider WANs has remained roughly constant over the years [11, 15, 19, 22, 33, 38].

What (more) can we do to avoid such outages? To answer this question, we analyzed the root cause of all major outages in a large cloud provider's SDN WAN over the last five years.

Our analysis revealed that a root cause of *over one third* of these major outages is one that has received relatively little attention within the research community: *incorrect inputs to the SDN controller*. In fact, incorrect inputs are the single largest contributing root cause in the outages we analyzed. In all these outages, the SDN controller itself operates correctly, but is compromised because it receives inputs that do not accurately reflect the current state of the network. For example, in several of the outages we analyzed, the controller received an incomplete view of the current traffic demand, leading to sub-optimal routes, congestion, and packet drops. In other outages, the controller received an incorrect view of the topology, which caused it to overload the links it believed to be operational.

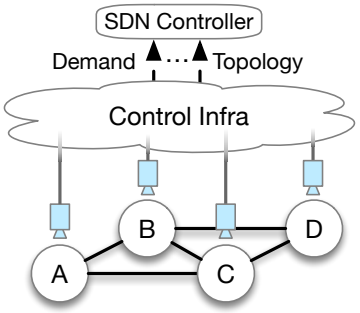

Figure 1: Simplified view of control infrastructure today.

This data is troubling not only due to the large fraction of outages but also because such inputs can render most existing techniques for ensuring availability (*e.g.,* verification, replication, testing, *etc.*) futile. This is because these techniques validate a system's output *assuming correct inputs*. As such, the guarantees they provide are rendered meaningless when the inputs to the system themselves are incorrect.

One may wonder how incorrect inputs are possible when the SDN controller reads network state directly from the routers. We delve into this question in §2 but, briefly, the answer lies in the complexity of production WANs. For one, the control infrastructure (Figure 1) that aggregates these inputs is complex, and spans dozens of services with millions of lines of code subject to frequent updates, making bugs unavoidable [12, 19, 20]. Additionally, incorrect inputs may result from faulty device-level telemetry signals that arise due to bugs in router hardware and software, both of which are complex in terms of lines of code and often blackboxes to network operators.

One might also wonder: what are operators currently doing to catch incorrect inputs? Our discussions with operators revealed that they do perform sanity checks on the inputs to their SDN controller. However, these checks are typically *static* in nature, and are often crafted to prevent *impossible* values of the input; *i.e.,* values that cannot possibly occur, such as topologies with more nodes than actually exist in the network. Operators also use static checks to catch *unlikely* inputs, using

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696874

heuristics based on historically correct values, past outage experiences, and so forth. Unfortunately, such checks are often ad-hoc and hard to manage; they accumulate over time across disparate parts of the system, and are local to their part of the system. Even worse, these heuristics are dangerous since they can result in false positive scenarios where atypical inputs are discarded despite correctly reflecting the state of the network at that time; *e.g.,* in a disaster scenario that impacts a large number of routers.

Perhaps more fundamentally, our analysis reveals that inputs are often incorrect not because they cannot possibly occur or are unlikely to occur, but because they are not *currently occurring*; *i.e.,* they do not reflect the *current* state of the network. Thus, detecting incorrect inputs requires comparing the inputs against the current network state. Existing approaches (based on static checks) are ill-suited for this because current state is dynamic and spans a wide range of operating points.

The goal of our paper is to shine a light on the problem of input validation in SDN networks. As our analysis indicates, this is an important problem that remains unsolved. We argue that input validation must be based on *dynamic* invariants that ensure an SDN controller's inputs reflect *current* network state.

Our proposed solution starts with the observation that dynamic validation requires an accurate view of current state, one that is resilient to bugs in even our low-level telemetry signals. To build confidence in these signals, we take advantage of the *symmetry* inherent to a network, which creates opportunities to corroborate a telemetry signal from one network location with independent signals taken from different vantage points in the network. For example, a trivial symmetry might be that the "bytes out" counter on one end of a link must match the "bytes in" counter on the other end, though more nuanced symmetries exist (§4).

Building on the above insights, we discuss the space of possible approaches, and propose our approach to input validation, Hodor, that proceeds in two steps. First, we exploit symmetry to validate device-level signals, yielding what we term a "hardened" set of router signals. Next, we dynamically generate invariants that validate the high-level inputs to an SDN controller against these foundational signals. Our early analysis suggests that this methodology could have averted the majority of the outages that stem from incorrect inputs in our dataset, though many open questions remain (§6).

2 Incorrect Inputs In Production SDN WANs

In an SDN WAN, the controller takes an abstract view of the network state as input. To create this input, the network's control infrastructure first collects *signals* from individual routers in the network (*e.g.,* link status, link metrics, demand, *etc.*), and then aggregates them (*e.g.,* by constructing a topology from individual link statuses), before passing them to the controller.

Since incorrect inputs caused over one third of all major outages over the past five years in the production SDN WAN we analyzed, we sought to better understand why these inputs occur; *i.e.,* why inputs to the SDN controller do not always reflect the current state of the network.

We observed that incorrect inputs arise in two scenarios: (1) when routers produce incorrect signals, and (2) when correct router signals are incorrectly processed and aggregated by the control infrastructure. We provide examples of both, along with brief descriptions of the outages they can lead to.

2.1 Incorrect Router Signals

Router signals can be of two types: those that reflect the state of the network (*e.g.,* telemetry signals), and those that reflect the intent of network operators (*e.g.,* a router's drain status). We find that both can be incorrect, and lead to outages. Though our outage analysis focused on an SDN WAN, below we also include examples of vendor router bugs experienced in a large cloud provider's traditional WAN that can impact the inputs to traffic management systems.

Telemetry Bugs. Telemetry data provided by routers is the primary way to gain visibility into network state. However, modern routers consist of not only complex hardware but also millions of lines of code of software, both of which provide a large surface area for bugs. For example, one observed bug in the router OS caused certain telemetry messages to be duplicated, with one of the two messages reporting (at random) that the number of packets received on the router's interfaces was zero. These messages led the control plane to interpret these interfaces as faulty and refrain from routing traffic through these otherwise functioning interfaces. Other reported bugs have included OS-level bugs that led to malformed telemetry responses, changes in telemetry format (*e.g.,* from `string` to `int`), delayed telemetry reporting, and incorrect QoS marking on telemetry packets, all of which led to incorrect or missing router signals. Note, all of these bugs manifested in production despite extensive pre-rollout testing, since the wide range of configurations, network loads, OS versions, *etc.* makes it infeasible to catch all such bugs before deployment [31].

Incorrect intent. Routers also report certain configurable signals that reflect the intent of network operators; for example, the "drain status" is used to mark a router undergoing maintenance or experiencing faulty behavior, and serves as a signal to the SDN controller to refrain from sending traffic to it. However, bugs in the router software can lead to this signal differing from the operator-intended view of the network. For example, in one outage, the interaction between a controller job restarting and a router marking itself as drained led to an inconsistent view of the drain status of the router's links, resulting in link congestion. Another outage was caused by an incorrect drain condition that had the opposite effect, and erroneously drained

a series of routers that were actually capable of carrying traffic, leading to congestion.

2.2 Incorrect Aggregation of Router Signals

Even when routers produce correct signals, the SDN controller can still receive a different view of the network.

Bugs in the control plane infrastructure. Bugs at any point in the processing infrastructure can cause correct router signals to be mutated or delayed. For example, in one outage, a new rollout of the topology instrumentation service introduced a bug that did not wait for all routers to provide their link statuses before stitching together the topology, thus providing the SDN controller with a partial view of the network causing severe congestion. In a second outage, a bug in a different instrumentation service caused it to misreport the liveness of particular links; this led to the SDN controller receiving a topology that had less bandwidth than was actually available, which caused sub-optimal traffic placement and local congestion. Finally, in a third outage, a router's (correct) drain signal was partially ignored by the topology instrumentation service; this led the service to incorrectly include the drained router's link capacities in the total available capacity which caused severe congestion.

External Input. The final reason we observed for incorrect inputs was because, unlike the simplified diagram shown in Figure 1, some portions of the input to the SDN controller may be collected *outside* of the network (*i.e.,* not from the routers). For instance, in the network we analyzed, demand information is collected from measurements at the end hosts rather than from routers directly (similar to [21]). This can lead to scenarios where the SDN controller can receive an incorrect view of such external inputs, despite everything in the network working correctly. For example, in one scenario, a new rollout of the demand instrumentation system introduced a bug that incorrectly aggregated demand at the end hosts. This caused the SDN controller to receive a partial view of the demand, which led to severe congestion and a major outage since the routes programmed by the controller did not take into account a significant fraction of the demand. In another major outage, the demand instrumentation service correctly measured demand, but this traffic was incorrectly throttled at the end hosts causing the measured demand to differ from the traffic that was allowed onto the network, and the SDN controller to make sub-optimal pathing decisions.

In summary, incorrect inputs to the SDN controller arise either due to bugs in router hardware and software that lead to incorrect input signals, or bugs in the control infrastructure that cause correct inputs to be aggregated incorrectly. Given the scale and complexity of production networks, detecting and eliminating these bugs in their entirety, using either testing or formal verification, is intractable [20]. Instead, we aim to *validate* the inputs, *i.e.,* provide confidence about whether the inputs correctly represent the current state of the network, *despite* any potential bugs in network components.

3 Approach

Given the complexity of production SDN WANs, building a system that guarantees correct inputs at all times is likely infeasible. Instead, we seek to *mitigate* the problem, by developing techniques that enable operators to quickly identify and fix incorrect inputs when they occur.

Our key insight is that signals in networked systems are heavily *correlated*; for example, a packet arriving at router X and leaving at router Y will be accounted for in the demand reported from X and to Y, as well as in incoming and outgoing interface counters at each router that the packet traverses. Thus, incorrect inputs will likely lead to *inconsistencies* in the set of signals obtained from the network (*e.g.,* discrepancies between a router's reported demand and the packet counts at adjacent interfaces). We seek to use these inconsistencies to detect the likelihood of an input being incorrect, and suggest fixes based on other correlated signals.

3.1 Design Space

The approach used to detect and fix incorrect inputs can vary based on the available knowledge of the system's design.

The most general approach assumes no prior knowledge about how different signals in the system are correlated—such as which values were derived from others (as in §2.2), which should be equal, or which should sum to others. Unsupervised learning techniques can be applied to discover this structure by analyzing historical system data, bundling all available data (including telemetry and control inputs) for each timestamp, and using methods like masked autoencoders [13] and symbolic regression [9] to identify relationships within these bundles that persist over time. However, these techniques may capture spurious relationships that, while true during the historical observation period, are not *fundamental* to the system's operation. For example, if the routers in a particular Point of Presence (POP) remain drained (up but not carrying traffic) during the historically observed period, unsupervised methods might infer that all interface counters in that POP should always be equal, which would no longer be accurate once the routers in the POP are undrained and counters become non-zero.

We propose a more specialized approach that incorporates constraints on how signals are correlated based on expert knowledge of the system's design. We leverage two aspects of our system knowledge. First, we partition the system data into low-level signals (*e.g.,* interface counters) which are close to the routers (§2.1) and high-level abstract data (*e.g.,* control inputs) which are aggregated from these signals (§2.2). This allows us to independently establish a trusted low-level view of the network before verifying that the high-level view aligns with it. Second, we manually define the expected relationships

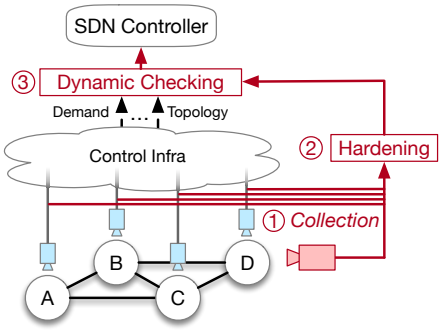

Figure 2: Our three-step approach to input validation.

within low-level signals and between the low-level network state and the high-level abstract data. We detail this approach below, and leave exploring a more general approach to future work.

3.2 Our Approach: Hodor

Our proposed approach to input validation (Hodor) consists of three steps (shown in Figure 2): ① *Collection*, which comprises reading (possibly incorrect) signals from individual routers to get a *comprehensive* view of the current network state, ② *Hardening*, which comprises detecting and (when possible) repairing incorrect signals to obtain a *correct* view of the current network state, and ③ *Dynamic checking*, which comprises checking that the inputs to the SDN controller are *consistent* with the current network state. Hodor is currently only an approach; we envision implementing it as an always-on system that continuously validates inputs to the SDN controller as it receives them.

Step ①: Collecting input signals. The goal of this step is to gather all the router signals that represent network-state inputs to the SDN controller. The key challenge here is to identify what signals are available, and whether they are relevant to the inputs to the SDN controller. To overcome this challenge, Hodor leverages the fact that network operators today maintain detailed network models [23, 25, 35], and use vendor-agnostic APIs [5, 26, 29] which provide detailed documentation about each available router signal. The relevant signals are chosen once at system design time, after which they are automatically collected, providing Hodor with a continuous and comprehensive view of the current network state.

Interestingly, we found that collecting a comprehensive view of router-level signals has another positive side effect: we discovered cases where inputs were incorrect because the system designers had defined their inputs on incorrect or insufficient signals. This was a *design* time bug and yet our approach (based on corroborating across multiple available signals) is able to detect the error. We discuss such a scenario in the context of a topology input in §4.2.

Step ②:Hardening input signals. This step answers the question of how to obtain a correct view of the network state, when bugs in routers may cause some of the signals collected in step ① to incorrectly reflect either the network state or operator intent, or be missing entirely (§2.1).

The key observation that helps us answer this question is that networks contain an inherent amount of *redundancy*, due to which a telemetry signal from one network location can often be corroborated against signals from other locations. Hodor leverages this redundancy to first *detect* signals that are likely to be incorrect (since they are inconsistent with signals observed from other locations), and then *repair* such signals (and missing signals), by using the values observed from other locations. We refer to this detection+repair process as "hardening."

The primary source of redundancy (R_1) that Hodor leverages to *detect* possibly incorrect signals is the symmetry across the two ends of a link: for example, the "bytes out" counter at one end of a link must approximately equal the "bytes in" counter at the other end,[1] and the "link status" reported at one end of the link must equal the status reported at the other end. When the two signals are not equal, Hodor flags both as "possibly incorrect" and proceeds to repair them.

In the repair step, our approach relies on more nuanced forms of redundancy to first obtain additional correlated observations, and then uses these observations to infer the true value for missing and possibly incorrect signals. The second source of redundancy (R_2) that Hodor relies on is flow conservation across all router ports; *i.e.,* the total number of "bytes in" across all interfaces of a router, must approximately equal the sum of the total number of "bytes out" and the "bytes dropped" at the router. When the counters for "bytes in" and "bytes out" at the two ends of the link are not approximately equal, Hodor checks if each counter maintains flow conservation with other counters at its router. Assuming an isolated incorrect counter, only one of the two counters would pass the above check, allowing Hodor to detect which counter is most likely incorrect and also repair its value.

The final two sources of redundancy that Hodor uses are: *alternative signals* (R_3) and *manufactured signals* (R_4). In R_3, Hodor uses the fact that routers provide different signals that can serve as possible alternatives; for example, to check whether a link is up, one can not only look at the reported status at both ends but also see if the "bytes in" and "bytes out" counters are zero at both ends. In R_4, Hodor relies on additional techniques that are deliberately introduced to create redundant signals. For example: using active neighbor probes to obtain additional signals as to whether a link is up or down. Hodor primarily uses R_3 and R_4 to increase confidence during the repair process. Since this involves inferring an unknown ground truth, the greater the number of signals, the higher the confidence that Hodor's inference is correct.

[1]The need for approximation arises due to discrepancies in the time window over which counters are measured and are accounted for as described in §4.

An open question is how much Hodor's hardening process efficacy may be diminished by correlated failures: for example, a bug in the vendor OS that causes multiple routers to report incorrect, but equal signal values. While such bugs are indeed possible, network operators already take several steps to reduce their impact including employing multiple vendors, and performing staged rollouts of router software updates. Using alternative signals adds another layer of protection through relying on different router-level signals: *e.g.,* a link status signal that is based on optical transceiver components vs. byte counts that are based on interface counters. That said, a detailed evaluation of hardening efficacy remains an open question that we are actively exploring.

Step ③:Dynamic checking. In the final step, Hodor checks that the inputs to the SDN controller are consistent with the hardened signals constructed in step ②. The checks are input-specific, and we provide examples for three inputs to the SDN controller in §4.

A natural question is: What course of action should Hodor take if the inputs are inconsistent with the hardened signals? This is a non-trivial question to answer, and one that arises even today with existing sanity checks in SDN WANs. We anticipate Hodor's validation checks to be integrated in a similar process to how existing checks are integrated today into alerting and management tools: for instance, Hodor can reject inputs that fail validation and fall back temporarily to the last input state, or trigger an alert for a reliability engineer to intervene. We leave this policy for operators to configure based on their operational model.

In summary, we propose a three-step approach to validate inputs. After ① collecting all relevant router signals, Hodor uses the redundancy inherent in networks to ② harden router signals, and mitigate the challenge posed by signals that incorrectly reflect network state or operator intent (§2.1). Finally, in step ③, Hodor checks inputs against the (hardened) current network state. Since this network state is obtained directly from routers, and is untouched by the control infrastructure's processing and aggregation logic, this step enables Hodor to detect bugs in both the control infrastructure and code that collects inputs from locations external to the network (§2.2).

One may ask whether Hodor is itself vulnerable to bugs and how this impacts its usefulness? While true, we note that this is a general concern for any validation/alarm system that pursues defense-in-depth; Hodor merely adds another layer of protection. Additionally, we believe that the surface area for bugs that Hodor introduces is relatively small since (unlike the existing SDN control infrastructure) Hodor does not process or aggregate signals but only reads and compares them. In addition, Hodor checks will likely be easier to maintain and manage since they do not rely on heuristics drawn from prior outage experiences. That said, we acknowledge that this question is best answered through production experience.

4 Design

We now describe how one might apply our approach for the three inputs to an SDN controller: the traffic demand (§4.1), the topology (§4.2), and the drain status of individual routers (§4.3). We focus on these three inputs since incorrect versions of these were the root cause of all large input-related outages we described in §2. Due to space constraints, we explain validation in detail for demand and provide high-level descriptions of our approach for topology and drain, leaving more detailed descriptions to follow-on work.

4.1 Demand

The high-level input for demand is matrix D, where D_{ij} represents the rate of traffic arriving at ingress router i that is destined for egress router j [36]. In the network we analyzed, D is computed from measurements at endhosts [21].

The router signals that we collect are interface counters that record the traffic rate sent/received on each interface measured over a few second rolling window. Note that these counter values are semantically different from D_{ij}: the latter captures an ingress-egress flow that travels multiple hops, while interface counters capture many flows on a single link. Nonetheless, their values are interdependent and we exploit this dependence for validation as described below.

As sketched in §3 our approach is to first harden the low-level interface counter values, and then check our high-level input D against these hardened counter values. We present each step in the validation process with a running example, shown in Figure 3, in which one counter on the A→B link reported a faulty erroneous measurements.

Hardening. For *detection*, we take advantage of the redundant signals that arise from link symmetry (R_1 from §3), comparing the outgoing interface count to the incoming interface count on each side of each link in the network. We deem any pairs of measurements missing or differing by more than some small hardening threshold τ_h[2] to be spurious and replace them with an unknown variable as we do not know which one is correct. The threshold allows for some natural difference due to the measurements being collected over a few second window. We average all other pairs of entries, producing a *flow vector* containing constants and variables for traffic volume on each link.

For *repair*, to correct the detected anomalies we take advantage of the flow conservation principle of networks (R_2 in §3):

$$\forall v \in V, \sum_{e \in E_{\text{in}}(v)} counter(e) = \sum_{e \in E_{\text{out}}(v)} counter(e) + dropped(v)$$

[2]This threshold depends on the network sampling frequency and traffic patterns. Based on production logs, we find 2% to be an appropriate threshold.

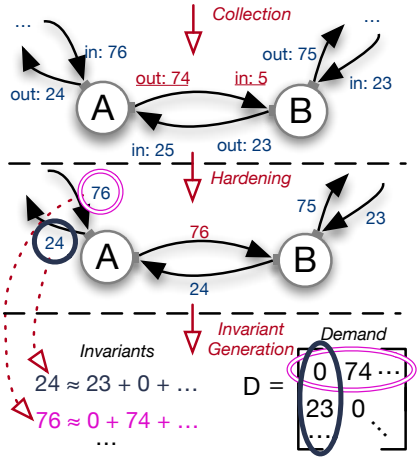

Figure 3: Simple example of demand validation, with external ingress/egress links shown on the left and right. The value on link A→B is detected to be spurious (red). Flow conservation at B finds the correct value is 76. The demand matrix contains demand from rows (A,B,...) to columns (A,B,...).

where $E_{\text{in/out}}(v)$ denotes the set of edges directed in/out of router v. Formulating this as a dot product between the incidence matrix M and $\vec{v}_{partial}$, we can solve for up to $|V|-1$ unknowns, the rank of M [4], to recover missing/corrupted values.

We show this in Figure 3, with the underlined counters for link A→B being detected as faulty due to their difference. We produce a graph with all other values hardened, then solve for the faulty link counter through flow conservation at B[3]:

$$\sum_{in} = \sum_{out} \quad \rightarrow \quad x+23=75+24 \quad \rightarrow \quad x=76$$

Dynamic Checking. Using the hardened set of interface counters, we can now produce invariant checks that relate our high-level demand matrix D to the per-interface counter values. Intuitively, the two are interdependent since the traffic in D_{ij} contributes to the per-interface counters at every edge along the path from i to j. We exploit this interdependence at each ingress and egress router. Concretely, we check that the total external[4] ingress rate at a router must equal the reported sum of demands from that router to all other routers. Likewise, total external egress at a router must equal the reported sum of demands from all other routers to this router. We show one of each for our running example in Figure 3.

The final result is $2v$ unique invariants, which is not enough to fully re-derive D (which contains v^2 entries) but does significantly constrain its range of acceptable values. To account for slightly differing numbers due to the hardening tolerance threshold, we choose an equality threshold τ_e, which accepts invariants within τ_e percent of equality.

Preliminary Evaluation. An initial analysis reveals that our approach would have successfully detected each case of incorrect demand inputs discussed in §2. In addition, as a sensitivity analysis, we tested the accuracy of our validation using demand matrices from the Abilene network [27] that we artificially "perturbed" to mimic buggy demand matrices. Our initial findings are encouraging: with $\tau_e = 0.02$, our approach detects 99.2% of perturbed matrices with two zeroed-out (missing) values out of 144, and 100% of perturbed matrices with three or more zeroed-out values. In ongoing work, we are evaluating the accuracy of our methods under a much wider range of scenarios.

4.2 Topology

The topology input to the SDN controller is a graph view of the network, derived from link status information reported by routers; this status signal is low level, reporting whether light is passing through and the link is administratively up.

A closer analysis of our scenarios involving incorrect topology inputs reveals that an input topology can be incorrect in two ways: either (1) the topology is misrepresenting the (non)existence of links, or (2) the reported data is *semantically* incorrect or insufficient, *e.g.,* a link is included in the topology as its interface status is up but traffic can't flow due to other reasons such as bugs in the dataplane, misconfigured ACLs, *etc.* Such bugs are introduced at *design* time, when the developer selected some router signals and defined how their inputs would be computed from the same. Thus, the role of hardening is both catching buggy or faulty signals, but also re-enforcing the intended semantic meaning of the signal.

To create a hardened and more comprehensive view of link status, we consider link status and interface counters (as mentioned in our discussion of demands) and we harden these values by comparing their values on each side of the link, and to each other. To provide further confidence, we further consider "manufacturing" redundancy by running limited active probes that periodically check that a link is up. A router can conduct these probes itself through a simple application running on the router [6, 19], similar to existing mechanisms used to check L2 link status [1]. Thus, our hardening process here leverages three of the redundancies discussed in §3: link symmetry, alternative signals, and manufactured signals. We leave out the full truth table of how to combine these signals, but note it can be adjusted based on risk tolerance of the operator. For example, if one side of a link reports up and the other down, but rate counters are all large and a probe succeeds, the link is likely up.

Once we have a hardened view of link status, dynamic checking is straightforward: we compare our hardened link status directly with the topology view at the SDN controller.

[3]We could equally have solved at A, where we would get a slightly differing solution due to rolling network telemetry. We could average solutions from all adjacent routers, or simply pick one.

[4]By external, we mean traffic leaving or entering the network domain, *e.g.,* to a datacenter Top-of-Rack (TOR) switch.

4.3 Drain Status

As discussed in §2, a router drain is a form of intent signal that marks a router to be avoided. It is applied in many scenarios including manually during outages to move traffic away from affected areas, pre-emptively to enable safe maintenance operations, and reactively through automation if faulty behavior is detected. There are many different drain mechanisms [7, 8, 16, 17] that differ slightly in their method and impact.

Drain is hard to validate as it is semantically overloaded, making it difficult to know where to seek redundancy. Looking across the above scenarios we find that, at a high level, an incorrect drain signal manifests as one of two cases: (1) drain is not marked when the router is supposed to be drained and cannot actually carry traffic, and (2) drain is marked when the router could actually still carry traffic. The first case is captured by the same mechanism as in §4.2, as interface counters and probes will be affected while interface status stays up.

The second case is harder as it requires verifying whether the router is justified in moving to a drain status. Probes may help to detect that the router can still carry traffic, but there are valid cases when a router claims to be drained despite being able to forward traffic, such as preemptive drains for maintenance.

Ultimately, the right approach might be to standardize the drain process for greater transparency through a mechanism that enables redundancy. One approach may be to attach reasons to drain labels, which can then be used to validate the drain. For example, a drain due to faulty neighbor connectivity can be validated by Hodor by checking the supposedly affected connection causing the drain. We could require all drains to be link drains, as link drains contain natural symmetry—both sides must agree that the link is drained. A node drain would then simply drain all links. An announced link drain can be validated by checking that the neighbor also announced a drain of that link. We leave a full exploration and evaluation of these approaches to future work.

5 Related Work

There is a vast literature on increasing network availability through testing [31], emulation [14, 24], and formal verification [3, 10, 18, 30, 37]. However, almost all such approaches aim to prove that an output is correct given a particular input (*e.g.,* configuration or values), and do not focus on validating the input itself.

The "Evolve or Die" paper [12] extracts learnings from 100+ outages at Google. While they do include examples of outages due to incorrect inputs, they do not report incorrect inputs as their leading cause of outages. We speculate this difference may be for a few reasons; *e.g.,* their analysis looked across both their SDN- and protocol-based WANs, is nearly a decade old, and some of their recommendations may now be common practice leading to new dominant causes.

Flexible Contracts for Resiliency [32] provides a language for reasoning about chains of assumptions that connect ground truth "signals" to higher level abstractions, similar to the approach we took in reasoning about the relationship between network signals and controller input. Adopting their formal methods into Hodor is an interesting direction for future work.

Anomaly detection [2, 28] approaches detecting outliers in input data through statistical analysis of a signal's past history. In contrast, we focus on whether a signal reflects the ground truth, and for that we look across signals for corroboration.

6 Conclusion and Future Directions

This paper makes the case for input validation as an open and important problem that merits attention. We sketched one possible approach to input validation, applied to the specific context of an SDN controller in the WAN. However, many open questions remain.

Hodor's approach. Open questions remain about the details of Hodor's design, including evaluating their efficacy and demonstrating a practical implementation of the same. Additionally, exploring a more general formulation of input validation as discussed in §3 may prove fruitful, and ought to be compared to the specialized approach we take in Hodor's design.

Designing more reliable networks. We are also curious if some of the techniques we identified might be useful to incorporate back into routers and the control infrastructure to help prevent the occurrence of incorrect inputs in the first place. For example, a router may exchange interface counters with its neighboring routers, in order to detect and self-correct anomalies in its reported data.

The broader design space and its applicability. What other approaches might one pursue for input validation? Are incorrect inputs a problem in other environments such as protocol-based WANs, datacenter fabrics, or CDN infrastructures? And would the approach we described be applicable to these environments? More generally, the problems discussed in this paper show up in all sorts of large systems, beyond just networked systems. For systems that naturally exhibit redundancy and interdependence due to the structure of the domain they operate on, the approach we propose could generalize to provide an inexpensive approach to increasing confidence in their inputs (compared to maintaining fully redundant control systems, as in existing critical systems [34]).

7 Acknowledgments

We thank our anonymous reviewers for their feedback and insightful comments. We also thank our colleagues and partners at Google for their support in exploring this idea and navigating Google data, including Bikash Koley, Subhasree Mandal, Hank Levy, David Culler, Doug Browning, and others. This work was supported by an NSF Graduate Fellowship.

References

[1] 2022. Ethernet CFM, Y.1731 Basic Concepts, Configuration, and Implementation. https://www.cisco.com/c/en/us/support/docs/asynchronous-transfer-mode-atm/operation-administration-maintenance-oam/117457-technote-cfm-00.html.

[2] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. 2016. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications* 60 (Jan. 2016), 19–31.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. *SIGPLAN Not.* 49, 1 (Jan. 2014), 113–126.

[4] R.B. Bapat. 2014. *Graphs and Matrices*. Springer London. https://books.google.com/books?id=LWCSBAAAQBAJ

[5] Carl Lebsack, Marcus Hines, Paul Borman, Anees Shaikh, Rob Shakir, Wen Bo Li, et al. 2018. gNMI - gRPC Network Management Interface. https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md.

[6] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: Building Switch Software at Scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 342–356.

[7] Cisco. 2005. *Uses of the Overload Bit with IS-IS*. Technical Report 24509. Cisco.

[8] Cisco. 2016. *IP Routing: OSPF Configuration Guide*. Cisco.

[9] Miles Cranmer. 2023. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl. *arXiv [astro-ph.IM]* (May 2023).

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

[11] Tony Fyler. 2023. Azure Outage Disconnects Thousands. https://techhq.com/2023/01/azure-outage-disconnects-thousands.

[12] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. https://dl.acm.org/doi/10.1145/2934872.2934891

[13] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2021. Masked Autoencoders Are Scalable Vision Learners. arXiv:2111.06377 [cs.CV] https://arxiv.org/abs/2111.06377

[14] Marcus Hines and Alex Masi. 2021. Kubernetes Network Emulator. https://github.com/openconfig/kne.

[15] Santosh Janardhan. 2021. Details About The October 4 Outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/. *Engineering at Meta* (Oct. 2021).

[16] Juniper. 2015. L2TP drain | Juniper Networks Pathfinder Feature Explorer — apps.juniper.net. https://apps.juniper.net/feature-explorer/feature-info.html?fKey=6874&fn=L2TP+drain.

[17] Juniper. 2020. BGP - Delay Route Advertisments - Wait for KRT Drain. https://apps.juniper.net/feature-explorer/feature-info.html?fKey=10427&fn=BGP+-+Delay+Route+Advertisments+-+Wait+for+KRT+Drain.

[18] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 113–126.

[19] Alexander Krentsel, Nitika Saran, Bikash Koley, Subhasree Mandal, Ashok Narayanan, Sylvia Ratnasamy, Ali Al-Shabibi, Anees Shaikh, Rob Shakir, Ankit Singla, and Hakim Weatherspoon. 2024. A Decentralized SDN Architecture for the WAN. In *Proceedings of the 2024 ACM SIGCOMM Conference*. https://doi.org/10.1145/3651890.3672257

[20] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. 2021. *Decentralized cloud wide-area network traffic engineering with BlastShield*. Technical Report MSR-TR-2021-31. Microsoft. https://www.microsoft.com/en-us/research/publication/decentralized-cloud-wide-area-network-traffic-engineering-with-blastshield/

[21] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[22] Frederic Lardinois. 2020. IBM Cloud suffers prolonged outage. *TechCrunch* (June 2020).

[23] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic Life Cycle Management of Network Configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks* (Budapest, Hungary) *(SelfDN 2018)*. Association for Computing Machinery, New York, NY, USA, 29–35.

[24] 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 *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 599–613. https://doi.org/10.1145/3132747.3132759

[25] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, Google Llc Xiaoxue Zhao, and Alibaba Group Inc. 2020. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA). USENIX Association, 403–418.

[26] OpenConfig Project. 2015. OpenConfig. https://www.openconfig.net/.

[27] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. 2010. SNDlib 1.0—Survivable Network Design Library. *Networks* 55, 3 (2010), 276–286. https://doi.org/10.1002/net.20371 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.20371

[28] Animesh Patcha and Jung-Min Park. 2007. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks* 51, 12 (2007), 3448–3470. https://doi.org/10.1016/j.comnet.2007.02.001

[29] Rob Shakir, Xiao Wang, Nathaniel Flath, et al. 2017. gRIBI - gRPC Routing Information Base Interface. https://github.com/openconfig/gribi.

[30] Ratul Mahajan Ryan Beckett. 2020. Capturing the state of research on network verification. https://netverify.fun/2-current-state-of-research/.

[31] Rob Sherwood, Jinghao Shi, Ying Zhang, Neil Spring, Srikanth Sundaresan, Jasmeet Bagga, Prathyusha Peddi, Vineela Kukkadapu, Rashmi Shrivastava, Manikantan KR, Pavan Patil, Srikrishna Gopu, Varun Varadan, Ethan Shi, Hany Morsy, Yuting Bu, Renjie Yang, Rasmus Jönsson, Wei Zhang, Jesus Jussepen Arredondo, Diana Saha, and Sean Choi. 2024. Netcastle: Network Infrastructure Testing At Scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 993–10008. https://www.usenix.org/conference/nsdi24/presentation/sherwood

[32] Michael Sievers and Azad M. Madni. 2014. A flexible contracts approach to system resiliency. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 1002–1007. https://doi.org/10.1109/SMC.2014.6974044

[33] Richard Speed. 2021. AWS runs into IT Problems. https://www.theregister.com/2021/12/15/aws_down.

[34] Ronald C Suich and Richard L Patterson. 1990. *How much redundancy: Some cost considerations, including examples for spacecraft systems*. Technical Report E-5592.

[35] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H Y Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 426–439.

[36] Paul Tune and Matthew Roughan. 2013. Internet Traffic Matrices: A Primer. https://matthew.roughan.info/papers/traffic_matrix_sigcomm.pdf.

[37] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 765–780.

[38] WIRED. 2019. The Catch-22 that Broke the Internet. https://arstechnica.com/information-technology/2019/06/the-catch-22-that-broke-the-internet/.