CrossCheck: Input Validation for WAN Control Systems

Alexander Krentsel[1,2] Rishabh Iyer[1] Isaac Keslassy[1,3] Bharath Modhipalli[2]
Sylvia Ratnasamy[1,2] Anees Shaikh[2] Rob Shakir[2]

[1] *UC Berkeley* [2] *Google* [3] *Technion*

Abstract

We present CrossCheck, a system that validates inputs to the Software-Defined Networking (SDN) controller in a Wide Area Network (WAN). By detecting incorrect inputs – often stemming from bugs in the SDN control infrastructure – CrossCheck alerts operators before they trigger network outages.

Our analysis at a large-scale WAN operator identifies invalid inputs as a leading cause of major outages, and we show how CrossCheck would have prevented those incidents. We deployed CrossCheck as a shadow validation system for four weeks in a production WAN, during which it accurately detected the single incident of invalid inputs that occurred while sustaining a 0% false positive rate under normal operation, hence imposing little additional burden on operators. In addition, we show through simulation that CrossCheck reliably detects a wide range of invalid inputs (e.g., detecting demand perturbations as small as 5% with 100% accuracy) and maintains a near-zero false positive rate for realistic levels of noisy, missing, or buggy telemetry data (e.g., sustaining zero false positives with up to 30% of corrupted telemetry data).

1 Introduction

Modern society is increasingly reliant on the Internet, yet state-of-the-art networks continue to exhibit regular outages. Major outages have been reported by virtually every large network operator [57, 18, 5, 38, 26], and a recent study shows that, despite sustained effort and investment, the frequency of such outages is not declining [33].

To better understand these outages, we studied the postmortem reports for all major outages in a large cloud WAN over a five-year period (2019–2024). Like many WANs operated by large cloud providers [22, 12, 25], this network employs an SDN-based control architecture in which a logically centralized software controller is responsible for routing and traffic engineering decisions. As such, we believe the findings from our study are broadly representative.

The most common root cause across the outages we analyzed—accounting for over one-third of all cases—was *incorrect inputs* to the SDN controller, i.e., inputs that *did not accurately reflect the current state of the network*. For example, in some outages, the SDN controller received an incomplete view of the current traffic demand, while in others it received an incorrect view of the current network topology. As expected, decisions based on incorrect inputs resulted in undesirable outcomes, including sub-optimal routes, congestion, link overloads, and packet loss. Informal conversations with practitioners at other cloud providers confirm that such incorrect inputs are not specific to the network we studied, but are also a common cause of outages elsewhere.

Why do incorrect inputs occur? The answer lies in the complexity of production WANs. State-of-the-art deployments incorporate control infrastructures spanning dozens of services and millions of lines of code, each subject to frequent updates. In addition, they rely on routers that are typically sourced from multiple vendors and comprise complex hardware and software stacks. Together, this complexity creates a large surface area for bugs, making it nearly inevitable that some controller inputs will be missing, stale, or incorrect.

Given that incorrect inputs are inevitable, we ask: *Can we build a system that detects when inputs to the SDN controller deviate from the current network state?* Here, inputs refer to higher-level, aggregated information such as traffic demand matrices or topology views, while current state is defined in terms of low-level dataplane signals exposed by routers, such as interface byte counters, link status indicators, and forwarding entries. Such a system would allow operators to detect input errors before their effects manifest, thereby avoiding a large class of outages.

To be useful, such an input validation system must satisfy two key requirements. First, it must continuously validate inputs in real time, at the timescales at which SDN control decisions are made. This requirement is motivated by our analysis of postmortem reports, which revealed that incorrect inputs typically arise not because they are syntactically invalid or impossible, but because they are inconsistent with the *cur-*

rent state of the network. Second, the system must maintain a near-zero False Positive Rate (FPR), *i.e.,* not raise an alert when the inputs are correct, while achieving a high True Positive Rate (TPR), *i.e.,* reliably raise an alert when the inputs are incorrect. We prioritize minimizing the FPR because, in a typical WAN, inputs are correct for the vast majority of time. Thus, even a modest FPR (e.g., 1%) would produce an unacceptable number of spurious alerts, as controller decisions are executed frequently (typically on the order of minutes). This would undermine operator trust and limit adoption.

Meeting these requirements in a production-scale WAN presents several challenges. First, measurements in real-world networks are inherently noisy, and the system must reliably distinguish benign noise from input errors to avoid false positives. Second, as noted earlier, even the low-level router signals used to represent current state may be incorrect due to router bugs. This introduces a seemingly recursive problem since the system must validate inputs against reference signals that may themselves be incorrect. Finally, implementing such a validation system in practice is non-trivial. The system must operate in real time, remain decoupled from the control plane to avoid shared failure modes, and be sufficiently simple to minimize the risk of bugs in the validator itself.

We present *CrossCheck*, an input validation system that can reliably detect incorrect inputs to an SDN controller even in the presence of noisy or faulty router signals. CrossCheck builds on two key insights. First, flow conservation in a network yields multiple, redundant measurements that should remain *consistent* during normal operation. For example, the `bytes_out` counter on one end of a link should match the `bytes_in` on the other end, and the traffic demand reported between a WAN ingress router and an egress one should be reflected in the interface counters of all intermediate routers. Second, any inconsistencies between these measurements reveal both the *presence* and the *nature* of these errors. Specifically, faulty or noisy router signals appear as *local* anomalies, such as a few counters reporting inconsistent values; whereas incorrect controller inputs, such as an incorrect demand matrix, produce *global* inconsistencies visible across all routers along the affected paths. CrossCheck leverages this asymmetry to distinguish incorrect inputs from noisy or faulty router signals, which enables it to achieve a near-zero FPR while preserving a high TPR.

To reduce the likelihood of bugs in the validation process, CrossCheck adopts an architecture that is decoupled from the general SDN control infrastructure. In CrossCheck, inputs to the controller and router signals are continuously streamed into dedicated databases, and the validation logic is implemented as a simple, stateless process that meets the timing requirements. Unlike current SDN control infrastructures, which are optimized for high performance and availability, CrossCheck's architecture is deliberately *lean*, thereby reducing the likelihood of new or correlated bugs.

We evaluated CrossCheck by running it as a shadow system for 4 weeks in the production WAN we analyzed, and by stress testing it in simulation across multiple topologies and error scenarios. In the production shadow deployment, CrossCheck incurred a 0% FPR and correctly detected the only incorrect input that is known to have occurred during the deployment period. In simulation, CrossCheck reliably detected incorrect inputs, and was able to identify *all* scenarios in which demand estimates were perturbed by 5% or more. CrossCheck also proved resilient to faulty telemetry, maintaining an FPR of 0% even when up to 30% of router signals were missing or corrupted. Finally, and perhaps most importantly, CrossCheck's accuracy improves exponentially with network size, since larger networks provide more interdependent signals, which suggests that it is especially suited for production WANs. Because CrossCheck only relies on fundamental network invariants, we also argue that it is more amenable to generalization.

2 Background and Motivation

Over the past two decades, both traditional ISPs and modern hyperscalers have adopted SDN-based control architectures to manage their WANs [22, 12, 25, 14]. At the core of these SDN systems is a logically centralized controller that runs a traffic engineering (TE) algorithm to compute capacity-aware paths [22, 25, 23, 12]. However, implementing TE in a global WAN entails far more than running a software process on a single datacenter server. In practice, it requires a variety of software services deployed across a global footprint of servers [22, 25]. These servers run a variety of services for topology discovery, demand estimation, switch programming, configuring policy, software upgrades, and more. Some of these services run on standard data center servers managed by a cluster management system [63, 58, 21, 59] while edge controllers may run on special servers that are co-located with routers and run a specialized orchestration platform [15, 7, 33].

In addition, because each of these components — hardware and software alike — is on the critical path for high availability, they are engineered accordingly with redundancy, replication, consensus, *etc.*, becoming fairly complex systems in their own right. For example, both edge and central controllers are commonly replicated across distinct hardware and geo-diverse data centers, running Paxos for consistency and failover [20]. Taken together, these systems represent dozens of non-trivial microservices and millions of lines of code. Prior work has highlighted the complexity of this control infrastructure and the inevitability of bugs and errors within the same [33, 35, 20, 55].

As noted earlier, we obtained sanitized diagnostic data from a major cloud provider covering all major outages over the last five years. Our analysis shows that more than one-third of these outages stemmed from incorrect inputs to the TE controller. To better understand how such errors occur, we examined individual outages in detail. In what follows, we first review the inputs to a TE controller, and then summarize

our findings on how incorrect inputs arise and why standard best practices for input validation often fall short.

2.1 TE Inputs

In an SDN WAN, the inputs to the TE controller consist of two key pieces of information: (i) **traffic demand**, which is a matrix D, where D_{ij} denotes the aggregate rate of traffic entering ingress router i and destined for egress router j [62]; and (ii) **topology**, which includes physical connectivity (*i.e.,* which links are present) and link capacity (since partial cuts on bundled links can result in reduced but non-zero capacity).

These inputs are typically computed from a few different data sources. In multiple networks that we are aware of, D is computed from measurements at *end hosts* [36, 22], while topology information is pieced together using network information in the *control plane* (*e.g.,* topology models [44], routers marked as undergoing maintenance, *etc.*) as well as link status signals read from individual *routers*. Information from these various sources is then aggregated as it travels from hosts and routers through the control hierarchy before being passed up to the TE controller.

2.2 How Do Incorrect Inputs Occur?

Our analysis revealed that incorrect inputs can occur in either of the above inputs, and generally arise due to the three reasons we discuss below.

1) Incorrect inputs from external sources (e.g., end hosts). As mentioned, inputs such as demand may be collected from sources *outside* of the network (*i.e.,* not from the routers). This can lead to scenarios where the SDN controller receives an incorrect view of demand, despite everything in the network working correctly. For example, in one scenario in our dataset, 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. 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 TE controller to make suboptimal path selection decisions.

2) Incorrect router signals. Telemetry data provided by routers is one source of information that goes into computing the high-level topology input. In addition, we are interested in using router signals for validation, as representing ground truth for the network's current state. However, router telemetry is itself not a flawless signal. Modern routers consist of complex hardware and millions of lines of code, both of which provide a large surface area for bugs. For example, we observed a scenario in which a bug in the router operating system 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, incorrectly removing them from the topology input to the TE controller and leading to unnecessary congestion in the network. Additional examples included router 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.

3) Bugs in the control plane infrastructure. Bugs at any point in the processing infrastructure can cause correct data, whether host measurements or 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 topology. In a second outage, a bug in a different instrumentation service caused it to misreport the liveness of particular links, leading once again to a partial topology view.

2.3 How Are Incorrect Inputs Detected Today?

Our discussions with operators indicate that, as one may expect, they perform sanity checks on inputs to try to catch incorrect inputs. However, these checks are typically *static* and designed primarily to prevent impossible values, *i.e.,* inputs that cannot occur, such as topologies claiming more nodes than actually exist in the network. Operators also apply static checks to flag unlikely inputs, relying on heuristics derived from historical values, past outages, and other experience. Unfortunately, these checks are often ad-hoc, defined in response to specific incidents rather than through a systematic process that considers inputs and network state holistically, as we aim to do. Operators further noted that these checks are difficult to maintain: they accumulate over time and must be updated continually to reflect evolving practices. More critically, such heuristics are risky, producing false positives where atypical but valid inputs are discarded—for example, during a disaster that affects many routers simultaneously. Ultimately, the limitations of static checks are evident in the fact that they failed to detect the incorrect inputs that led to the outages we analyzed. For example, despite the existence of a static validation check that no single metro region was missing all routers, an outage was caused by a topology input bug where a large portion (but not all) of routers were dropped from many metros.

2.4 Bad Input Causes a Bad Day

We walk through one situation in detail to illustrate how bad-input-caused outages arise despite the best intentions of network operators. In one outage we observed [11], an update was rolled out to a subset of the regional jobs that read interface telemetry directly from routers in the network, and aggregate it into abstract views of the topology. This update introduced a race condition bug, which when triggered caused the jobs to not properly wait for responses from

all routers before constructing and passing up an abstract connectivity graph from the routers in their region. As a result, these connectivity graphs appeared to be missing a significant portion of the actually available capacity.

The top-most abstract aggregator job received these partially incomplete sub-aggregations, and stitched them to produce a final global abstract topology missing roughly a third of actual available capacity, which was then given to the SDN controller to use for pathing and traffic placement.

The SDN controller produced paths using the available capacity in the abstract topology, fitting as much demand as possible but was unable to fit all demand because of the lack of capacity. This led to traffic throttling and congestion.

The SDN controller did have some static checks in place to catch badly malformed inputs; for example, checking that the provided topology was not empty, or that no single region was empty. These did not prevent the input from proceeding because the topology was not empty, and all regions had at least some capacity.

We note that in this scenario, the SDN controller's solver produced correct results given its inputs, i.e., the paths produced by the SDN controller were the best possible paths for a topology missing the capacity stated in the abstract topology. Rather, the issue was that the input given to the controller did not accurately reflect reality.

In summary, incorrect inputs to the SDN controller arise for a variety of reasons, and the current practice of using static sanity checks is both insufficient and risky. Given the scale and complexity of production networks, eliminating these bugs entirely via extensive testing or formal verification is also intractable [35]. Motivated by this state of affairs, we propose *input validation* as a complementary strategy that acts at runtime to contain the impact of such bugs.

3 CrossCheck Overview

We now present an overview of CrossCheck, a system that continuously validates inputs to the SDN controller against the current *network state*, which is defined in terms of low-level dataplane signals exposed by routers. We begin by describing the overall architecture of CrossCheck (§3.1), then introduce the specific router signals it collects and explain why they are suitable for validation (§3.2). Finally, we provide intuition for how CrossCheck exploits the inherent redundancy in network behavior to construct a reliable view of the current state, even in the presence of noisy or faulty signals (§3.3). We describe the design of CrossCheck in detail in §4.

3.1 System Architecture

Fig. 1 presents an architectural overview of CrossCheck, which validates SDN controller inputs in three stages. First, in the ① *collection* step, router signals and controller inputs are continuously streamed into a centralized backend database.

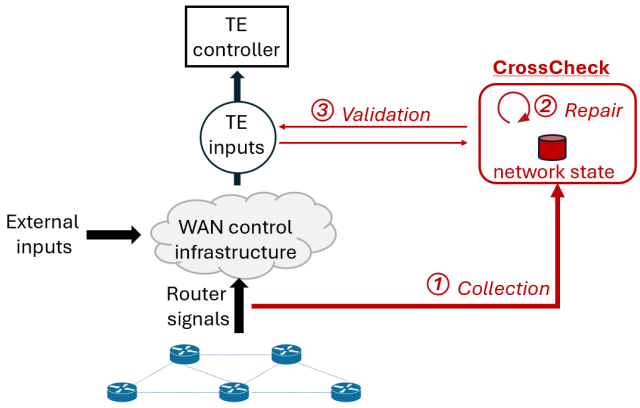


Figure 1: CrossCheck high-level system design.

Second, in the ② *repair* step, CrossCheck constructs a reliable network-wide view of the current state from these collected signals. Third, in the ③ *validation* step, CrossCheck checks whether the controller inputs are consistent with this reconstructed state and classifies them as either *correct* or *incorrect*. We adopt this binary decision model for simplicity, though CrossCheck could be easily extended to additionally abstain if it detects that too many router signals are missing or corrupt for it to reach a confident verdict.

CrossCheck's architecture is motivated by two considerations. First, decoupling CrossCheck from the SDN control plane and streaming data into a dedicated backend allows the validation logic to remain simple, and isolated from shared failure modes. Second, repairing the network state holistically, enables CrossCheck to exploit the inherent redundancy in router signals (*e.g.,* due to flow conservation across the two ends of a link or at a router) to identify and tolerate faulty signals and measurement noise. Together, these choices enable CrossCheck to maintain a near-zero False Positive Rate (FPR), *i.e.,* avoid flagging correct inputs, while achieving a high True Positive Rate (TPR), *i.e.,* reliably detecting incorrect ones.

3.2 Collected Router Signals

CrossCheck collects three types of router signals. We chose these signals because they are directly relevant to the inputs being validated (namely, demand and topology), and because they are consistently available across router platforms via standardized APIs [10, 53]. Table 1 summarizes the collected signals, which we explain below.

(1) Link status indicators. For each directed link l from router X to Y, CrossCheck collects four link status indicators. First, the *physical status* of the link at each router, denoted l^X_{phy} and l^Y_{phy}, which reflects physical-layer conditions such as the detection of optical signals. Second, the *link-layer status* at each router, denoted l^X_{link} and l^Y_{link}, which indicates whether each router considers the link active, based on the successful exchange of heartbeat messages with the remote endpoint. CrossCheck does not require additional heartbeat traffic for

Type	Signal		Notation
	Name	Location	
Link status indicators	Physical status	egress	l^X_{phy}
		ingress	l^Y_{phy}
	Link-layer status	egress	l^X_{link}
		ingress	l^Y_{link}
Link counters	Counters	transmit	l^X_{out}
		receive	l^Y_{in}
Forwarding entries	Entries	router X	$F_X(\rightarrow l_{demand})$

Table 1: Collected router signals and their notations.

these signals; instead, it reuses the link status computed by existing protocols such as BFD [28, 8], which are already deployed on modern routers.

(2) Link counters. CrossCheck collects hardware counters from each router interface that track the cumulative number of bytes sent and received. For a directed link from X to Y, CrossCheck uses the transmit counter at X (l^X_{out}) and the receive counter at Y (l^Y_{in}). To compute traffic rates, CrossCheck samples these counters periodically and derives per-interval rates from the difference in values and timestamps. For simplicity, we continue to refer to these as l^X_{out} and l^Y_{in}.

(3) Forwarding entries. Finally, CrossCheck collects the forwarding table F_X from each router X. At ingress routers, F_X specifies how incoming traffic is encapsulated into tunnels (via encapsulation rules), and at transit routers, it determines how each tunnel is forwarded through the network. By combining forwarding entries across routers, CrossCheck reconstructs the path of each tunnel and estimates the load contributed by each demand $D_{I,J}$ on every link. We denote this estimated load on link l as l_{demand}.

We highlight two important properties of the signals described above. First, among the seven link signals only two—namely the physical statuses l^X_{phy} and l^Y_{phy}—are used to compute the inputs to the SDN controller (specifically topology), and only one—namely the estimated load l_{demand}—is dependent on the inputs to the SDN controller, since l_{demand} is derived using the demand input. The remaining four signals are thus completely independent of the controller inputs, hence offering a distinct view of the network state that can be used to validate those inputs. That said, CrossCheck does not rely solely on these independent signals, since excluding the input-dependent signals would increase vulnerability to faults in the others (§6).

Second, each of the signals is collected from a different component within a router, which reduces (though does not eliminate) the possibility that they are all simultaneously buggy. In our analysis of all major outages logged over a five-year period, we found zero occurrences of simultaneous bugs across even two of the above three classes. CrossCheck leverages this independence when validating the network topology, which we describe in §4.3.

3.3 Network Invariants

If the router signals collected by CrossCheck were always accurate, validating SDN controller inputs would be straightforward. For instance, one could validate demand by comparing the router-measured load on a link, $l_{router} = (l^X_{out} + l^Y_{in})/2$, with the load inferred from the demand input, l_{demand}. Similarly, one could validate topology by checking whether the link-layer status indicators l^X_{link} and l^Y_{link} agree with the controller's view of the link's state.

In practice, however, router signals are often noisy, missing, or even incorrect (§2). Hence, CrossCheck first detects and corrects such faults in a process we call *repair*. This repair step enables CrossCheck to reconstruct an accurate view of the network's state, which it then uses to validate controller inputs while maintaining a low false positive rate.

CrossCheck's repair process leverages the observation that router signals are *strongly correlated* across the network. As a result, faults in one signal often produce inconsistencies with others.

Concretely, CrossCheck leverages four network invariants that must hold in any correct network to repair faulty router signals. We now define these invariants for a directed link l from router X to router Y, and describe how CrossCheck performs the repair in the next section.

(1) Link invariants. These reflect the requirement that (i) both ends of a link must agree on its operational status, in both the physical and link layers:

$$l^X_{phy} = l^Y_{phy} = l^X_{link} = l^Y_{link}, \tag{1}$$

and (ii) the link must preserve flow conservation:

$$l^X_{out} = l^Y_{in} \tag{2}$$

(2) Router invariants. Every router must also obey flow conservation, *i.e.,* the total incoming traffic should equal the total outgoing traffic:

$$\sum_{\forall l} l^X_{in} = \sum_{\forall l} l^X_{out} \tag{3}$$

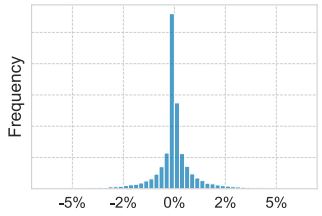
(3) Path invariants. The traffic demand estimated using the forwarding tables should match the load observed on the corresponding link.

$$l_{demand} = l^X_{out} = l^Y_{in} \tag{4}$$

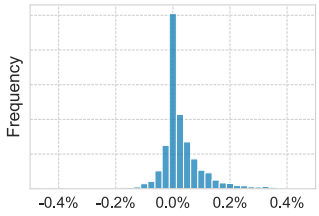
Why invariants may not hold. While each of the above invariants is intuitive and is expected to hold in an ideal network, many regular real-world effects can conspire to break these invariants in practice, even without the occurrence of any bug. Invariants may not hold due to effects in (1) the data plane, including packet propagation, queuing and processing delays, since these invariants assume zero delays, as well as packet drops, since the path invariant assumes

Invariant Status	Fraction of total time
Agreement	99.98%
Disagreement	0.02%

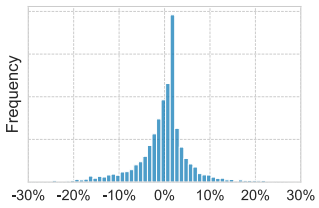
(a) Link status invariant



(b) Link invariant



(c) Router invariant



(d) Path invariant

Figure 2: Measured imbalance in our network invariants for a large production WAN. For (b)-(d), 0% implies that the equality holds perfectly.

that all demand arrives at all routers along its path; (2) the control plane, since paths are not updated atomically across all routers; and (3) the measurement layer, for example due to measurement noise and loosely-synchronized clocks.

Measured real-world invariants. To confirm the degree to which these invariants hold in practice in the face of the above real-world effects, we analyzed router telemetry data from two large production WANs *A* and *B* with O(100) nodes and O(1000) nodes respectively, over five-minute windows in a two-week period. Fig. 2 depicts the results for WAN *A*. The results for WAN *B* are similar and presented in Appendix A.

We first measured the percentage of link status reports in which our status-agreement invariant holds, *i.e.,* the routers on either side of a link agree on whether the link is up or not. We see disagreement on the link status only 0.02% of the time and hence our invariant holds 99.98% of the time (Fig. 2(a)).

We also check the degree to which our load-related invariants (*i.e.,* Eqs. (2)–(4)) hold in practice. Fig. 2(b) examines the link invariant derived from flow conservation: we plot the PDF of the difference between the link counters on each end of a link (Eq. 2) and see that they differ by less than 4% for 95% of links. Similarly, Fig. 2(c) examines our router invariant, plotting the PDF of the difference between incoming *vs.* outgoing rates at a router. We see that our router invariant (Eq. 3) holds within 0.21% for 95% of routers. This is the tightest load invariant, because all measurements are local to a router. Nonetheless, minor differences remain due to measurement offsets, packets in flight through the router, and dropped packets. Finally, Fig. 2(d) examines our path invariant, plotting the PDF for the difference between l_{demand} and the average of l^X_{out} and l^Y_{in} (Eq. 4). We see that the path invariant holds fairly well, but with a larger tail: for 75% of links the difference is within 5.6%, with a difference of 15.3% at the 95-percentile. The main reason is that paths may be updated during the measurement period and collecting path information is not synchronized (across routers or with path updates), introducing discrepancies between the load estimated from demand inputs *vs.* the measured link loads.

Since our load-related invariants do not hold exactly, we define a threshold $\mathcal{N}$ and say that an invariant holds if the relevant measurements are within $\mathcal{N}$ of each other. In our experiments, we set $\mathcal{N} = 5\,\%$, which corresponds to the 96.5th, 100th and 71.7th percentiles in Eqs. (2) to (4) respectively. CrossCheck will then use this threshold $\mathcal{N}$ in the repair algorithm to determine when two estimates for a link load can be deemed as equivalent (§4.1).

4 CrossCheck Repair and Validation

We now describe CrossCheck's repair and validation steps in detail. Recall that the two inputs to be validated are *demand* and *topology*. Since both rely on a common repair step to construct a reliable view of the current network state, we begin by describing this shared repair process (§4.1). We then present the validation logic, which is different for each input (§4.2 and §4.3).

4.1 Repair

The goal of CrossCheck's repair algorithm is to derive a reliable value for the traffic load on each link (denoted l_{final}). CrossCheck then uses the derived l_{final} not only to validate the demand (by comparing it against l_{demand}), but also as an additional source of ground truth for validating the topology, since $l_{final} > 0$ if and only if the link is up. For simplicity, we describe our repair algorithm informally in this section; the full algorithm is shown in Appendix D.

Voting. The general strategy of our repair algorithm is to accumulate multiple estimates for a link's load, each obtained from different sources. We call these estimates *votes*, and rely on a simple majority vote to select l_{final}. Given a link l from X to Y, the collected router signals already give us three votes: l_{demand}, l^X_{out}, and l^Y_{in}. Including l_{demand} as a vote might seem non-intuitive given it is computed from the high-level demand inputs. However, this is a deliberate choice we make to avoid false positives in the face of incorrect router telemetry: *i.e.,* because l_{demand} is independent of router counters, it can "vote against" the estimates derived from buggy counter values. Our evaluation in §6.3 confirms the value of this design choice. However, simply taking the majority of these 3 votes does not give us sufficient resilience to correlated bugs. For example, a counter bug at both routers X and Y could impact l^X_{out} and l^Y_{in} in the same way.

Deriving additional votes from router invariants. CrossCheck derives additional votes based on the observation that signals across routers are heavily correlated via the router invariants. For example, consider the link $X \rightarrow Y$ in Fig. 3.

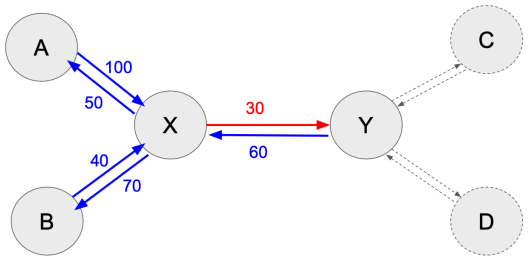


Figure 3: Example network with a faulty router signal (in red).

Applying our router invariant at X (*i.e.,* incoming load equal outgoing load) using the blue links, we can estimate the load on $X \rightarrow Y$ as $(100+40+60)-(50+70) = 80$. Doing the same at Y would yield an additional estimate. Thus we can obtain 2 additional votes by taking into account the load on the *neighboring* links of $X \rightarrow Y$.

However, this raises the question: When computing the router invariant at each router, which vote should CrossCheck use for each link? Specifically, while the example in Fig. 3 shows a single value for the load on each neighboring link, in practice, it can choose three baseline estimates for each neighbor link. For example, the load on the link m from $A \rightarrow X$ could be estimated using m^A_{out} or m^X_{in} or m_{demand}.

Ideally, we would like to leverage all possible votes, since each of them comes from a distinct source and thus increases our tolerance to correlated bugs across multiple routers. For example, considering m^A_{out} when we apply router invariants at X could allow us to catch bugs that impact X but not A. More generally, if a sufficient number of the routers neighboring X and Y are bug-free and we start counting their votes, we could obtain a majority vote that overrides the incorrect counter estimates provided by X and Y (as later formalized in Thm. 1).

Multiple voting rounds. Taking into account all the votes for each link rapidly leads to a state explosion problem when we consider all possible combinations for each link and router. For instance, just in the small example of Fig. 3, there are three estimates for each of the five blue links that surround router X, yielding $3^5 = 243$ tuples of estimates. To address this, the CrossCheck repair algorithm runs many *rounds* of voting. In each round, it randomly picks one of the three candidate values for each link l (l_{demand}, l^X_{out}, and l^Y_{in}), and then applies the router invariant at each router in the resulting topology. Thus, after N rounds of voting, each router X has N votes for the load at each of its incident directed links. We then merge the votes in agreement, *i.e.,* those within the previously-defined threshold of $\mathcal{N} = 5\%$, taking their average as their representative. The value cluster with the maximum number of votes serves as X's estimate for that link's load as derived from router invariants, which we denote as l^X_{rtr}. In addition, we compute a corresponding weight score (denoted w^x_{rtr}) which is simply the fraction of the N votes that were cast for l^X_{rtr}.

Consolidating votes: from five to one. Our procedure so far results in five estimates for each link l's load: l^X_{out}, l^Y_{in}, l_{demand}, l^X_{rtr} and l^Y_{rtr}. We assign a weight of 1.0 to the first three estimates, and weights w^x_{rtr} and w^y_{rtr} as above to the last two. [1] Our next step is to derive a single final estimate (*i.e.,* l_{final}) from this set. For this, we cluster votes that are within our noise threshold, summing the weights of any clustered values, and then pick the estimate with the largest cumulative weight as our *tentative* final estimate for l's load, with the corresponding cumulative weight as our confidence score for this estimate. However, before proceeding to finalize a link's load estimate, we undertake one final step as described below.

Gossip before finalizing. Our procedure so far uses a one-shot vote approach, which leaves us vulnerable to local pockets of correlated bugs. For example, in Fig. 3, repair fails if routers X, Y, A, and B all suffer from the same counter bug, since only the l_{demand} estimates would be exempt from the bug, and these would be out-voted by the counter-derived estimates.

Loosely inspired by gossip algorithms, we want to give a chance for values with high confidence to propagate and influence other values. We achieve this by using the repair algorithm *iteratively*. At each iteration, CrossCheck only finalizes the link with the highest confidence, and then repeats the above process of running multiple rounds, tallying votes, and so forth. Namely, CrossCheck picks the link $\tilde{l}$ with the highest confidence score and assigns its (previously tentative) final estimate to $\tilde{l}_{final}$. Crucially, once finalized, a link's estimate is fixed in all subsequent iterations. Thus for a topology with L links, CrossCheck runs the above repair process L times, finalizing one link's estimate each time, until it arrives at a value of l_{final} for each link l.

At this point, CrossCheck has a reliable (though not necessarily perfect) value for the load on each link. We next describe how it uses this value to validate the inputs to the SDN controller, beginning with the demand input, followed by the topology input.

4.2 Validating Demand

When validating demand, CrossCheck needs to answer the question: Given the repaired link loads (l_{final}), how can we determine whether the demand matrix provided to the SDN controller is correct, in a manner that ensures a near-zero FPR and a high TPR?

A natural approach is to apply the updated path invariant, *i.e.,* check whether the demand-induced load l_{demand} matches the repaired link load l_{final}, and alert the operator when the two differ. However, this raises the question: is any single violation of this invariant sufficient evidence of a faulty demand input? Or could such violations arise from the larger noise in path invariants and/or residual errors in router telemetry that were not fully corrected during the repair step?

To correctly identify incorrect inputs, CrossCheck

[1] We choose 1.0 for the first three estimates based on the intuition that they rely on direct measures at l's incident routers X and Y; our evaluation shows that this simple choice performs sufficiently well (§6).

leverages the observation that *the pattern of inconsistencies between l_{demand} and l_{final} can reveal the underlying cause*. Specifically, incorrect demand inputs tend to induce *widespread* violations of the path invariant: for example, a single erroneous entry D_{IJ} in the demand matrix will manifest as inconsistencies along all links traversed by traffic between routers I and J. In contrast, noise and residual faults in the telemetry of one router produce *localized* inconsistencies, typically confined to the set of its adjacent links.

To check for widespread violations of the path invariant, CrossCheck proceeds in two steps. First, given some threshold τ, it considers that a path invariant holds at a link l when its imbalance ($l_{final} - l_{demand}$) falls within τ. Then, in order to make a top-level validation decision, CrossCheck relies on a validation cutoff Γ, classifying a demand input as correct when the fraction of links for which the path invariant holds is above Γ, *i.e.,* when it does not encounter widespread violations. On the other hand, a buggy demand will encounter few links where its imbalance falls within τ, and therefore its fraction of satisfied links is expected to be below Γ, *i.e.,* it will be classified as incorrect.

At each new WAN, CrossCheck sets τ and Γ after an initial calibration phase, where it collects telemetry data and input demand matrices during a known-good period (*e.g.,* after an operator confirmation of stability). Given the collected path imbalance distribution during this calibration phase, τ is automatically set at the 75th percentile of this distribution.[2] Then, for each recorded time interval, CrossCheck applies the repair procedure, computes the number of links satisfying the path invariant, and records the resulting fraction. To maintain a near-zero FPR, CrossCheck sets Γ to just below the minimum value observed across this calibration window. For example, in WAN *A*, CrossCheck first went through a two-week calibration period. It set $\tau = 5.588\,\%$ as the 75th percentile of path-invariant imbalance, and $\Gamma = 71.4\,\%$.

At runtime, CrossCheck validates demand inputs by comparing the observed consistency rate to this threshold. If the fraction of links on which the path invariant holds exceeds Γ, the input is classified as *correct*; otherwise, it is flagged as *incorrect*. Algorithm 1 summarizes this procedure in pseudocode.

Configuring hyperparameters. CrossCheck's validation algorithm relies on four key hyperparameters mentioned above that balance detection sensitivity against false positive rate. We summarize and compare them here:

(1) The noise threshold $\mathcal{N}$ accommodates inherent measurement noise in production networks by defining when two load estimates are considered equivalent during the repair process. The network operator sets it based on an empirical analysis of the distribution tail of the difference in counters, which we set conservatively to be 5% given the link and router distributions shown in Figs. 2(c) and 2(d).

(2) The number N of voting rounds in the repair algorithm determines how many random combinations of link estimates are explored when applying router invariants, with more rounds providing greater resilience to correlated failures at the cost of increased computational overhead. We found $N = 20$ to be effective for our network. The optimal N is correlated with average node degree.

(3) The imbalance threshold τ determines the acceptable discrepancy between demand-induced load estimates and repaired link loads for individual links, effectively controlling the amount of noise in top-level invariants that our system is tolerant of. It is set by observing the distribution of invariant imbalances over a system observation period, which we show for our network in Fig. 2(d).

(4) The validation cutoff Γ applies to the final top-level validation decision, specifying what fraction of interfaces must satisfy the path invariant for the entire demand input to be classified as correct. This threshold is critical for maintaining near-zero false positive rates and is likewise calibrated during an initial observation period by setting it to be just below the minimum consistency rate observed during known-good operation.

Algorithm 1: Demand validation.

```
satisfied_count = 0;
foreach link l do
    if percent_diff(l.demand, l.final) ≤ τ then
        satisfied_count += 1;
return satisfied_count / num(links) > Γ;
```

4.3 Validating Topology

Validating the topology input is comparatively straightforward, as CrossCheck can rely on *five* independent signals to directly corroborate the status of each link. These signals are considered independent because they originate from distinct subsystems within the router.

For a directed link l from router X to router Y, CrossCheck uses the following signals: l^X_{phy}, l^Y_{phy}, l^X_{link}, l^Y_{link}, and whether $l_{final} > 0$. The first two represent the physical-layer status as reported by X and Y, respectively. The next two reflect the link-layer status as determined by heartbeat protocols like BFD. The fifth signal, l_{final}, reflects observed traffic on the link after the repair step and is computed using counters from across the network, making it independent of the others.

Given these independent sources, CrossCheck applies a simple majority vote across the five signals to determine the operational status of each link. In our evaluation, this approach successfully resolved the rare 0.02% of cases where the link status invariant was violated (Fig. 2(a)), and detected all instances of incorrect topology inputs encountered in our log of outages.

[2] This is a heuristic: a large percentile would accept large imbalances and therefore miss bugs that affect small demand volumes, while a small percentile may be too sensitive to counter perturbations.

4.4 Analyzing CrossCheck's Algorithms

Repair guarantees. Assume that the input demands and paths are correct, and that we set the threshold $\mathcal{N}$ high enough to capture regular noise when consolidating votes in the repair algorithm. Then the CrossCheck repair algorithm can provably detect and repair any corrupted counters at an arbitrary single (directed) link if counters at other links are fine. We show the full proof in Appendix B.

Theorem 1. *CrossCheck is guaranteed to detect and repair any corrupted counters when corruption is restricted to an arbitrary single link.*

For example, in Fig. 3, even if the egress counter of X and the ingress counter of Y at link $X \to Y$ are *both* corrupted (and the rest of the network only suffers from normal noise), CrossCheck is guaranteed to repair link $X \to Y$.

Scalability of demand validation with network size. Evaluating the consistency between l_{demand} and l_{final} at a network-wide level ensures that CrossCheck becomes more accurate as the network grows; *i.e.,* the FPR decreases, and the TPR increases with larger networks.

Formally, if we assume that the path-invariant imbalance follows an i.i.d. distribution across all links of all networks when given some healthy (resp., faulty) input demands, then:

Theorem 2. *As the number of links n increases, both FPR and $(1-TPR)$ converge to zero exponentially fast.*

The full proof is presented in Appendix C; we summarize the key intuition here.

Why the FPR decreases with network size. As the number of links grows, the distribution of invariant deviations observed across the network more closely matches the true underlying distribution. This happens because, with more samples (i.e., links), the overall pattern becomes more reliable. The DKWM inequality [43, 51] formalizes this intuition by bounding how far the observed distribution can deviate from the true one, and it shows that such deviations become exponentially smaller as the number of links increases. Thus, in large networks, the fraction of links satisfying the invariant stabilizes, allowing CrossCheck to set conservative thresholds that minimize false positives with high confidence.

Why TPR increases with network size. Larger networks naturally involve more traffic demand entries and more paths per demand. Hence, even a small number of incorrect demand entries affect a large portion of the network state.

Consider a 150-router network with 100 border routers and 50 transit routers. This setup yields $100^2 = 10{,}000$ distinct demand entries. If each demand traverses 5 routers on average and is multipath-routed over 4 disjoint paths, then a single demand results in roughly $5 \times 2 \times 4 = 40$ counter readings (ingress and egress at each router). Across all demands, this amounts to $10{,}000 \times 40 = 400{,}000$ total signal contributions.

Now, assume 5% of demand entries are incorrect. These affect $0.05 \cdot 400{,}000 = 20{,}000$ signal contributions. Spread over an estimated $150 \cdot 10 = 1{,}500$ router interfaces (assuming average degree 5 and 2 counters per port), each interface experiences over 13 error-induced discrepancies on average.

Thus, this analysis shows that even a modest fraction of incorrect demand inputs creates a dense pattern of invariant violations, allowing CrossCheck to detect such faults with high confidence in large networks.

5 Implementation

We built a prototype implementation of CrossCheck that performs input validation, using the architecture shown in Fig. 1. The CrossCheck prototype consists of two main components. The lower half is network-specific and handles the collection and storage of router signals, while the upper half is network-agnostic and performs repair and validation over these signals when provided as input. CrossCheck has been deployed as an experimental shadow system at a large cloud provider WAN (WAN *A*), where it has been continuously validating the inputs to their TE controller over a four-week period.

Collecting and storing router signals. The lower half of CrossCheck is responsible for collecting and storing the raw telemetry required for validation. This component is intentionally kept simple and does not perform any of the aggregation performed by the control infrastructure, to reduce the chances of bugs. Its implementation is specific to the telemetry interfaces available in the target network.

In WAN *A*, CrossCheck collects all telemetry via gNMI [10, 48], a standardized telemetry API supported across major router operating systems. CrossCheck subscribes to physical and link-layer status event updates for each link, and samples byte counters every 10 seconds per interface [47], emitted as a stream of (*timestamp*, *total-bytes-in/out*) tuples from which transmit and receive rates are derived. Forwarding paths are reconstructed using abstract forwarding entries retrieved from routers, which specify how traffic is split and forwarded across tunnels. All telemetry is stored in an in-house, in-memory time-series database that is similar to systems such as TimescaleDB [61], InfluxDB [24], and Monarch [2].

A natural question is whether such a "flat" system that performs no aggregation can scale. To answer this we calculate the write rate for a moderately-large network, storing roughly 10 metrics every 10 seconds from $O(10{,}000)$ interfaces. This rate of $O(10{,}000)$ writes per second is well within the peak performance of up to 2.4 million inserts per second shown for open-source time-series databases in prior work [60], and is easily accommodated by our in-house database system.

Finally, since raw router counters report monotonically increasing byte totals timestamped at each sample, CrossCheck must convert these into byte-per-second rates for use in repair and validation. Our timeseries database supports a query

language, which CrossCheck uses to issue a short query – just five lines – that aggregates interface counters into bundles and computes rate estimates over time. Occasionally, counters may reset due to hardware overflows or router restarts; CrossCheck explicitly detects and excludes such intervals from rate computation to avoid introducing spurious artifacts.

Repair and validation. The upper half of CrossCheck is responsible for consuming telemetry and executing the repair and validation procedures described in §4. This component is designed to be general-purpose: it interfaces with any backend time-series database via a pluggable API that abstracts over telemetry retrieval.

The implementation is lightweight. The core repair and validation logic is written in Python and comprises approximately 250 and 100 lines of code, respectively, supplemented by a few hundred lines of orchestration code for scheduling runs and logging outputs. CrossCheck exposes a simple `validate(demand, topology)` API, which takes a demand matrix and topology as input and returns a binary validation decision.

Deployment as a shadow system. The WAN is mission-critical in cloud infrastructure and hence we are unable to directly integrate our research prototype into the critical path of the production TE system. However, we are able to run CrossCheck as a "shadow" system (that is off the critical path) by leveraging an independent storage replica of the live TE database used in WAN *A*. Thus CrossCheck runs on real-time inputs (demand and topology) and telemetry from WAN *A* but our validation decision output is not integrated into the production TE system.

6 Evaluation

We evaluate CrossCheck through a combination of deployment as a shadow system in a WAN (§6.1) and simulation (§6.2). The former allows us to test CrossCheck under realistic conditions, while the latter allows us to explore a range of possible error scenarios we did not encounter in production. Our metrics are the TPR and FPR, as defined in §1, and our goal is to ensure a near-zero FPR, while maximizing the TPR.

6.1 Shadow Deployment Evaluation

Here, we seek to answer the following questions: (1) Will the noise inherent to production data trigger false positives? (2) Will CrossCheck be able to detect incorrect inputs in the real world? and (3) What are CrossCheck's performance characteristics when run on production data volumes?

To answer these questions, we deployed CrossCheck as an experimental shadow system in a cloud WAN (WAN *A*), where it has been continuously validating the inputs to their TE controller over a four-week period. In porting CrossCheck from the lab to production, we discovered a handful of practical changes that were necessary. First, we adjusted our equality accounting for headers as the load estimates derived from demands were systematically 2% lower than interface counters. We discovered that this is because, depending on the vendor, counters on some routers include bytes due to packet headers while our demand inputs do not. Second, we observed that for datacenter-facing interfaces, we needed to include *hairpinned* traffic—*i.e.,* traffic not routed on the WAN, but coming up and back down from the datacenter—when computing l_{demand}; again this traffic is not accounted for in the demand inputs but appears in router counters.

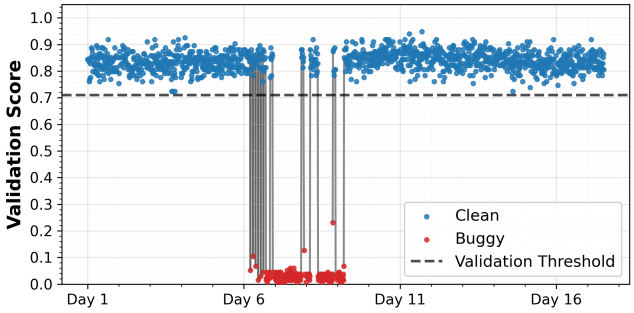


Figure 4: Shadow-system validation on live production data from a large production WAN *A*.

FPR and TPR in production. Over the 4 weeks that CrossCheck was deployed, we saw **zero false positives**, showing that our CrossCheck system is resilient to the inherent noise in real-world demand and telemetry data.

Additionally, our system **correctly detected a rare instance of incorrect input data**: during a short period, the database holding demand was affected by a bug introduced in a new code release, which caused it to double-count the demand measured at the end hosts. As a result, all demands in this replica were doubled for most of a period of 3 days before the issue was detected manually and rolled back.

Fig. 4 demonstrates the output of CrossCheck's validation during the period that the input data was incorrect. We can see that incorrect inputs cause a steep drop in validation scores when demands are found to be inconsistent with the state of the network derived from the router signals.

We highlight two takeaways from this incident. First, CrossCheck was able to detect an incorrect input where existing static checks failed to do so. Second, although the incorrect inputs did not affect our production TE controller (since this was a separate storage replica), they did affect some lower-importance capacity planning systems.[3] Moreover, this class of bug might equally have impacted the production DB, leading to a real outage. Indeed, similar bugs triggered some of the outages we reported in §2.

We also tested CrossCheck retrospectively over a separate storage replica that stores topology health data which is consumed by a network health sentry service that monitors and drains unhealthy links. Reading the production topology

[3] Capacity planning is done over a longer time-window and less frequently, which is how the bug went undetected for multiple days.

inputs and raw telemetry data, CrossCheck was able to successfully catch a scenario of invalid topology input that caused all healthy links at a router to be drained, leading to congestion causing an outage. If fully deployed, CrossCheck could have detected and prevented this outage, as it was able to detect that the links were up and healthy through the topology repair and validation process described in §4.3.

System performance. Since CrossCheck must continuously validate inputs in real time, at the timescales at which SDN control decisions are made (typically minutes [22, 33]), its runtime is equally important to its utility. We found that CrossCheck's runtime is within 10 seconds on our large WAN inputs. Router signals are typically available within the database within O(1 s) of when they are produced by routers. We benchmarked the queries that produce aggregated and average counter values, and observed that they take ∼56 ms to run. The largest component of runtime is repair, which takes ∼9.1 s to run, while validation takes O(100 ms). Put together, our system runs well within our target window and adds minimal latency overhead to typical TE iteration times. For latency-sensitive use-cases, to prevent delaying the control system, CrossCheck can be run on inputs in *parallel* to the control system beginning any computation necessary for decision making, with the validation result checked before it finishes, allowing the control system to proceed with any live action. We believe a more optimized repair algorithm implementation may be able to further speed up runtime.

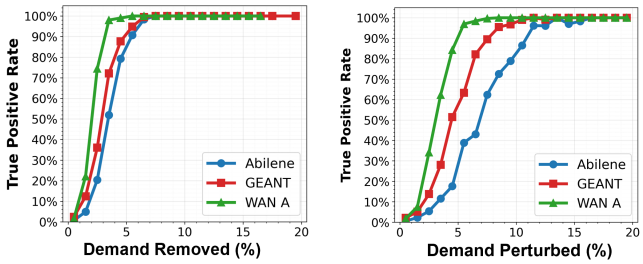
6.2 Simulation-Based Evaluation

We test our system's performance across a wider range of potential operating conditions in simulation. We use production traces of demands, paths, and topology from WAN *A* with $O(100)$ routers and $O(1000)$ links[4], as well as the open-source demands and topology from the Abilene (12 routers, 54 links) and GÉANT datasets (22 routers, 116 links) [49, 27].[5] We assume all-pairs shortest-path routing for Abilene and GÉANT.

Simulated telemetry. Once we have the demand, topology, and path information, we can use the path invariants – *i.e.,* derive per-link loads by tracing the demands along their paths – to calculate the counter values for all interfaces in the network. However, these counter values are idealized since they ignore the noise we observe in practice. We thus *add noise* to these idealized counter values. The amount of noise we add is carefully selected to match the real-world link-invariant, node-invariant and path-invariant noise distributions we reported in Fig. 2. We refer the reader to Appendix E for the precise methodology we follow. The resultant counter values are our baseline for a *healthy* network that has noise but no telemetry or demand bugs.

[4]We report uni-directional links, including those for network ingress/egress.

[5]We choose these two topologies as they are, to our knowledge, the largest public topologies that release production demand measurements.



(a) Demand removals. **(b)** Demand removals and additions.

Figure 5: CrossCheck's TPR with buggy demands. The x-axis displays the sum of the absolute values of the demand changes as a percentage of the total demand.

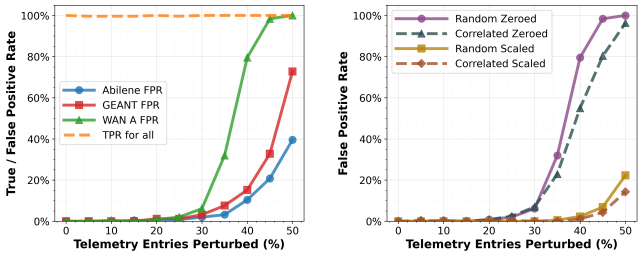
Dataset sizes. Our snapshots of WAN *A* are taken every 15 minutes over a four-week period in Spring 2025, totaling 2,000 snapshots, and our GÉANT and Abilene datasets comprise the first 4,000 snapshots of each.

Modeling buggy demands/telemetry. We model bugs by applying a range of random perturbations. When fuzzing demand that is taken as input to TE, we first pick a random number between 5% and 45% of demand entries to perturb, and then pick a range at random from 5%-15%, 15%-25%, 25%-35%, 35%-45% from which to sample the amount of traffic to change in each affected demand entry. We stop at 45% as larger changes are easier to detect. At each entry, demand is either always removed to model bugs that omit demand, or removed and added with equal probabilities to model bugs causing stale demands. To plot, we then compute the total percent of absolute demand changed in the snapshot. For telemetry, we use the same method, and specify the percent of counters and the perturbation range used.

6.2.1 Results

Is CrossCheck effective at catching perturbations to demand? We sweep a range of demand perturbations and plot the resulting TPR as a function of the absolute change in demand. Fig. 5 depicts the results. Fig. 5(a) shows our results when modeling bugs that only removes demands. We see that for such errors, *CrossCheck is very effective at detecting demand perturbations.* It can detect 74% of the 2-3% perturbations, and 100% of the 5%+ perturbations.

Fig. 5(b) shows our results when modeling bugs with stale demands (*i.e.,* in which demand elements are scaled up or down with equal probability). The outages we have observed only involved demands scaled in one direction, but we include stale ones as a harder test case. We note that this is a particularly challenging scenario, as we are effectively shifting demand between flows, keeping the total demand constant on average. The results are only slightly worse for WAN *A* compared to only removing demand and therefore CrossCheck does more than just checking the total demand. We see that very small networks (Abilene) are affected more greatly, as there is less path diversity, so additions to one



(a) Random counter zeroing. **(b)** Four telemetry perturbations applied to WAN *A*.

Figure 6: CrossCheck's FPR with buggy telemetry.

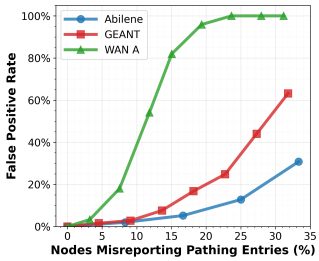


Figure 7: CrossCheck's FPR with buggy path information.

demand cancel out subtractions from another when sharing the same links. Nonetheless, TPR remains high for larger changes in demand volumes, *e.g.,* approaching 90% when 10% of demands are perturbed.

As detailed in §4.2, these strong TPR results are due to the fact that demand perturbations are reflected in many invariant checks (at each interface that demand traverses). Thus, a few demand changes can push enough validation invariant checks outside of the equality threshold, causing CrossCheck to detect the perturbed demand. We observe empirically that sensitivity increases with network size, as expected from Thm. 2.

Is CrossCheck resilient to buggy counter telemetry? Buggy telemetry can cause validation invariants to fail even though the demand is correct, leading to false positives. CrossCheck's repair process is designed to catch and rectify telemetry bugs, enabling demand validation to proceed safely. Fig. 6(a) shows the performance as we perturb telemetry by zeroing an increasing percentage of counter values, simulating dropped or missing telemetry, which is the most common form of telemetry corruption. We choose zeroing as such errors are harder to repair than random scaling, because if both sides of a link are zeroed, their agreement is difficult to overcome to recover the original value.

Our results show that *CrossCheck is very resilient to buggy counter telemetry*, able to withstand up to 30% of telemetry counters being zeroed before experiencing false positives. We observe the (pleasing) trend that with larger topologies, CrossCheck becomes increasingly resilient to telemetry perturbations. We also show in Fig. 6(a) that CrossCheck maintains 100% TPR in the face of telemetry perturbations in *all* cases (orange line, assuming 10% of demand volume randomly removed). This is because the set of buggy demand inputs that make up the TPR already causes enough invariants to fail to be detected; perturbing telemetry randomly only causes *more* invariants to fail.

Do correlated failures challenge CrossCheck's effectiveness? Telemetry perturbations often occur due to router-level bugs, and thus may be correlated, meaning they affect all of the interfaces of a particular router at once. Fig. 6(b) shows four classes of telemetry failures, comparing correlated and fully random zeroing and scaling (by 25%-75%). We see that *CrossCheck's repair process fully recovers from random and correlated failures* with up to 25% of the telemetry having errors, while the FPR starts increasing for higher percentages.

Note that correlated failures do not seem to significantly increase FPR. This is because they do not increase the probability that both sides of a link agree about an erroneous link value, *e.g.,* with zeroing, or that a single side is wrong. Moreover, the repair algorithm was designed to poll other sources of information for the node-invariant vote, and thus to address correlated failures.

Is CrossCheck resilient to buggy paths? The other telemetry CrossCheck relies on is forwarding entries that define paths. Though very rare, a router can possibly fail to correctly report some or all of its forwarding entries due to either a hardware or software fault on the router. We evaluate a particularly pessimistic node failure mode where each affected router reports not having *any* forwarding entries, sweeping the percent of routers in Fig. 7. Our results show that the FPR stays at zero up until we go above 4% of nodes experiencing faults. Such failures typically only affect one router, hence in practice are typically well below the point at which CrossCheck begins to experience false positives. Further, such bugs are easily detected, and in such cases the best strategy would be to skip validation.

6.3 Factor analysis

We now analyze whether CrossCheck's repair step – and each of its components – contributes to ensuring its low FPR.

Impact of repair on demand validation. We want to evaluate the impact of the different steps of the repair algorithm on CrossCheck's demand validation. For simplicity, we show results for just the GÉANT network in this section. We start by assuming a router bug affects 30% of the counters (denoted *random*) or all counters in 30% of the routers (denoted *correlated*), and they are either zeroed or scaled down by a random factor chosen uniformly at random in the range [25%,75%].

We test scenarios designed to highlight the contribution of certain key design choices: (i) granting a vote to our demand-induced estimate, l_{demand}, (ii) using multiple rounds of voting, (iii) the repair step in its entirety. Fig. 8 illustrates the resulting FPR. We see that running the CrossCheck validation without repair yields poor FPR results of over 90% in all cases. If we run a single round of the repair algorithm without the vote from l_{demand}, FPR only goes down slightly. However, if we run a single round with all five votes, FPR

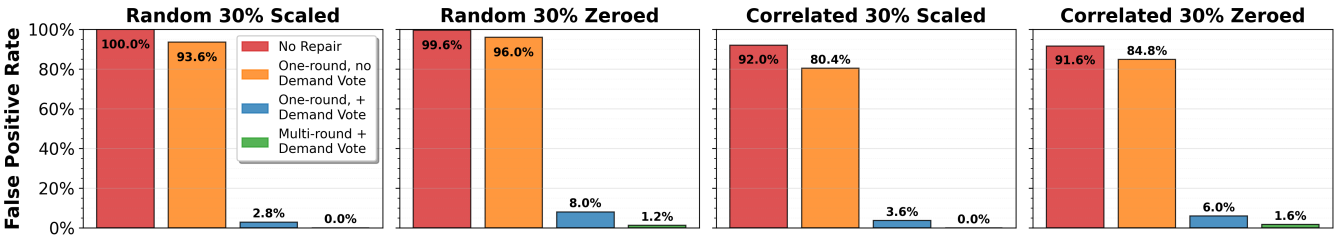


Figure 8: Factor analysis of the impact of our repair algorithm's design choices on CrossCheck's FPR in GÉANT.

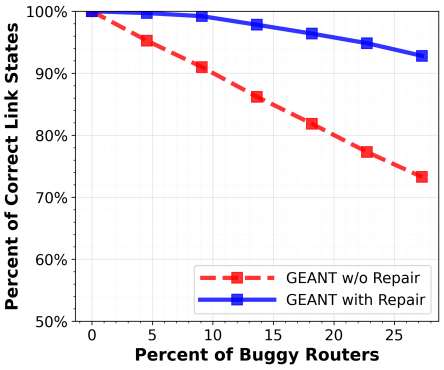


Figure 9: Effectiveness of topology repair in GÉANT.

goes down significantly. Thus, the additional demand-based tie-breaking estimate brings the most significant contribution to FPR. FPR is slightly lower for scaling bugs because they are easier to repair than zeroed counters. With scaling bugs, if two routers that share a link are affected, the two router estimates are different and therefore the error is easier to repair. If both are zeroed, then they agree on the same value and it is harder to make them abandon this value. Finally, the multiple voting rounds eliminate most of the remaining FPR by bringing global information in cases where router counters are stuck in local minima. All cases bring down the FPR to under 2%. Thus, the overall impact on FPR is dramatic, since the network-wide view magnifies it.

Impact of CrossCheck's repair on topology validation. We now want to study the effectiveness of CrossCheck repair for *topology validation* (§4.3). We consider the GÉANT WAN, and assume a worst case scenario that for any buggy router, *all* telemetry (physical status, link layer status, *and* counters) for all interfaces are buggy, reporting status down and counter 0, even though the links are actually functioning. We run our repair algorithm and plot the percentage of links we can correctly identify to be up, both before and after repair as we increase the number of buggy routers. Fig. 9 shows that repair manages to solve some 2/3 of the incorrect link states, even when over 1/4 of the routers are buggy.

In summary, our evaluation of CrossCheck across both a production deployment and simulation demonstrates that our system is both highly effective at detecting even small (5%) demand perturbations, and resilient to telemetry causing false positives. Our shadow deployment shows that our design is scalable to production data rates, stable enough to not produce false positives due to real-world noise, yet sensitive enough to catch real-world bugs.

7 Related Work

Prior Workshop Publication. This paper extends our earlier workshop publication [32], which introduced the motivation for input validation in SDN systems and outlined a high-level sketch of a potential solution. In contrast, the design, repair and validation algorithms presented in this paper are entirely new. Our prototype implementation and evaluation are also new and were not part of the workshop paper.

Formal verification. The research community has looked extensively to formal verification as a tool to catch or prevent network issues [54, 4, 64, 1, 6]. Some works, such as Batfish [17] and HSA [29], have enjoyed real-world adoption and commercial success. Adjacent approaches include using network emulation directly [40, 19], integrating verification and emulation [34], or general network testing techniques [55]. All of these methods aim to enforce some properties on the output behavior given some inputs, without considering the correctness of those inputs. CrossCheck is complementary to these efforts as it checks at runtime that the inputs correctly reflect reality, which we show is a significant category of outage.

Output validation. Some existing tooling considers validating network behavior by comparing actual to expected behavior. For instance, the Jingubang [39] module of Hoyan [66] simulates the expected traffic load of the non-SDN Alibaba WAN, then compares with monitored load. However, almost all such approaches aim to prove that an output is correct given a particular input and do not focus on validating the input itself.

Reasoning language. Flexible Contracts for Resiliency [56] 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 our system is an interesting direction for future work.

Outage statistics. The "Evolve or Die" paper [20] extracts

lessons from 100+ outages at Google. Although they do include examples of outages due to incorrect inputs, they do not report incorrect inputs as their leading cause of outages. We speculate that this difference may be due to a few reasons; *e.g.,* their analysis looked at 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. Likewise, SkyNet [65] recently reported on the leading network failure root causes at the non-SDN Alibaba WAN.

Anomaly detection. Anomaly detection [3, 50] can detect outliers in input data through statistical analysis of a signal's history. In contrast, we focus on whether a signal reflects the ground truth, and so look across signals for corroboration.

Statistical tools. In the validation step, we essentially want to check whether the distribution of the invariant imbalance is different from a typical distribution given clean input demands, and especially whether the imbalance values are higher. There are many two-sample statistical tests for checking if one distribution with non-negative values is stochastically larger than another, *e.g.,* the one-sided Kolmogorov-Smirnov test or the Anderson-Darling test [37]. Our validation scheme that focuses on the imbalance distribution tail is designed to reduce the sensitivity to bugs in the router counters. Early evaluations indicate that it is competitive with other tests, but more evaluations are needed.

8 Conclusion

We presented CrossCheck, a system that validates the inputs to the WAN SDN controller and alerts operators when these inputs appear incorrect. Running CrossCheck as a shadow validation system in a large WAN, we found that it caught an invalid input on live production data, while maintaining a 0% false positive rate on healthy inputs, thus avoiding false alarms to the operators. Finally, we showed through simulation that CrossCheck can catch up to 100% of demand perturbations over 5%, while being resilient to noisy telemetry with up to 30% of telemetry perturbations.

While our focus in this paper is on validating SDN inputs for TE, the methods we present can apply just as well to non-SDN TE such as RSVP-TE, where similar invariants to ours can be checked at each router on the flooded global network state. We further believe this class of input validation problem generalizes to a far wider range of control systems; while the particular invariants we selected for this paper were specific to our problem (*e.g.,* relating sums of demand values to interface counters), the methods used to find them were principled and can be applied more broadly to other classes of systems to validate their input, both for network control systems beyond TE such as link health monitoring, and more generally for control systems beyond networks such as building climate control or power management systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd Matthew Caesar for their insightful comments and helpful feedback. We appreciate early feedback from our colleagues at UC Berkeley, including Scott Shenker, Sarah McClure, Emily Marx, and others. We also thank our colleagues at Google, including Bikash Koley, Brent Stephens, Aniket Pednekar, Anurag Sharma, Hank Levy, David Culler, and others for their discussions and contributions to the development of CrossCheck. We particularly thank Sorin Constantinescu for his deep practical network telemetry knowledge and help in procuring live datasets. This work was partly supported by the Louis and Miriam Benjamin Chair in Computer-Communication Networks.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: fast multilayer network verification. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 201–220, USA, 2020. USENIX Association.

[2] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment*, 13(12):3181–3194, 2020.

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

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

[5] Brian Barrett. The catch-22 that broke the internet. https://arstechnica.com/information-technology/2019/06/the-catch-22-that-broke-the-internet/, June 2019.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William

Snow, and Guru Parulkar. Onos: towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Manav Bhatia, Guoyi Chen, Sami Boutros, Marc Binderberger, Jeffrey Haas, et al. Bidirectional forwarding detection (bfd) on link aggregation group (lag) interfaces. RFC 7130, February 2014.

[9] Emmanuel Candes and Benjamin Recht. Exact matrix completion via convex optimization. *Communications of the ACM*, 55(6):111–119, 2012.

[10] Carl Lebsack, Marcus Hines, Paul Borman, Anees Shaikh, Rob Shakir, Wen Bo Li, et al. gnmi - grpc network management interface. https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md, Jun 2018.

[11] Google Cloud. Incident report: Multiple services are being impacted globally within google cloud. Technical report, Google Cloud, February 2023.

[12] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, et al. EBB: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 346–359, 2023.

[13] David L Donoho, Arian Maleki, and Andrea Montanari. Message-passing algorithms for compressed sensing. *Proceedings of the National Academy of Sciences*, 106(45):18914–18919, 2009.

[14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: an intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

[15] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98. USENIX Association, April 2021.

[16] Mohammad H Firooz and Sumit Roy. Network tomography via compressed sensing. In *IEEE Globecom*, pages 1–5, 2010.

[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, pages 469–483, 2015.

[18] Tony Fyler. Azure outage disconnects thousands. https://techhq.com/2023/01/azure-outage-disconnects-thousands, January 2023. Accessed: 2024-1-30.

[19] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 167–181, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *ACM SIGCOMM*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.

[23] Chi-Yao Hong, Subhasree Mandal, Mohammad A. Alfares, Min Zhu, Rich Alimi, Kondapa Naidu Bollineni, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Jeffrey Liang, Kirill Mendelev, Steve Padgett, Faro Thomas Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jon Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *SIGCOMM'18*, 2018.

[24] Influx. InfluxDB Time Series Data Platform. https://www.influxdata.com/, 2025.

[25] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*, 2013.

[26] Santosh Janardhan. Details about the october 4 outage. *Engineering at Meta*, October 2021. Accessed: 2024-1-30.

[27] Piotr Jurkiewicz. Topohub: A repository of reference gabriel graph and real-world topologies for networking research. *SoftwareX*, 24:101540, 2023.

[28] Dave Katz and Dave Ward. Bidirectional forwarding detection (bfd). RFC 5880, June 2010.

[29] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, pages 113–126, 2012.

[30] Raghunandan H Keshavan, Andrea Montanari, and Sewoong Oh. Matrix completion from a few entries. *IEEE Transactions on Information Theory*, 56(6):2980–2998, 2010.

[31] Raghunandan H Keshavan, Andrea Montanari, and Sewoong Oh. Matrix completion from noisy entries. *Journal of Machine Learning Research*, 11:2057–2078, 2010.

[32] Alexander Krentsel, Rishabh Iyer, Isaac Keslassy, Sylvia Ratnasamy, Anees Shaikh, and Rob Shakir. The Case for Validating Inputs in Software-Defined WANs. In *ACM HotNets*, page 246–254, 2024.

[33] Alexander Krentsel, Nitika Saran, Bikash Koley, Subhasree Mandal, Ashok Narayanan, Sylvia Ratnasamy, Ali Al-Shabibi, Anees Shaikh, Rob Shakir, Ankit Singla, and Hakim Weatherspoon. A decentralized SDN architecture for the WAN. In *ACM SIGCOMM*, 2024.

[34] Alexander Krentsel, Oliver Ye, Anthony Tafoya, Xuqian Ma, Sylvia Ratnasamy, and Anees Shaikh. Towards accessible model-free verification. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, HotNets '25, page 210–217, New York, NY, USA, 2025. Association for Computing Machinery.

[35] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with Blastshield. Technical Report MSR-TR-2021-31, Microsoft, November 2021.

[36] 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. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, August 2015. Association for Computing Machinery.

[37] Paul Kvam, Brani Vidakovic, and Seong-joon Kim. *Nonparametric statistics with applications to science and engineering with R*. John Wiley & Sons, 2022.

[38] Frederic Lardinois. IBM cloud suffers prolonged outage. *TechCrunch*, June 2020.

[39] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, et al. Reasoning about network traffic load property at production scale. In *USENIX NSDI*, pages 1063–1082, 2024.

[40] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: A novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[42] Yi Lu and Balaji Prabhakar. Robust counting via counter braids: An error-resilient network measurement architecture. In *IEEE Infocom*, pages 522–530, 2009.

[43] Pascal Massart. The tight constant in the Dvoretzky-Kiefer-Wolfowitz inequality. *The Annals of Probability*, pages 1269–1283, 1990.

[44] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[45] Andrea Montanari. Estimating random variables from random sparse observations. *European Transactions on Telecommunications*, 19(4):385–403, 2008.

[46] Wolfgang Mulzer. Five proofs of chernoff's bound with applications. *arXiv preprint arXiv:1801.03365*, 2018.

[47] OpenConfig. openconfig-interfaces. https://openconfig.net/projects/models/schemadocs/yangdoc/openconfig-interfaces.html, 2025.

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

[49] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. Sndlib 1.0—survivable network design library. *Networks*, 55(3):276–286, 2010.

[50] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[51] Henry WJ Reeve. A short proof of the Dvoretzky–Kiefer–Wolfowitz–Massart inequality. *arXiv preprint arXiv:2403.16651*, 2024.

[52] Olivier Rioul and Patrick Solé. An information theoretic proof of the Chernoff-Hoeffding inequality. *Information Processing Letters*, page 106582, 2025.

[53] Rob Shakir, Xiao Wang, Nathaniel Flath, et al. gribi - grpc routing information base interface. https://github.com/openconfig/gribi, jul 2017.

[54] Ratul Mahajan Ryan Beckett. Capturing the state of research on network verification. https://netverify.fun/2-current-state-of-research/, April 2020. Accessed: 2022-10-9.

[55] 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. Netcastle: Network infrastructure testing at scale. In *USENIX NSDI*, pages 993–1008, Santa Clara, CA, April 2024. USENIX Association.

[56] Michael Sievers and Azad M. Madni. A flexible contracts approach to system resiliency. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1002–1007, 2014.

[57] Richard Speed. AWS runs into IT problems. https://www.theregister.com/2021/12/15/aws_down, December 2021. Accessed: 2024-1-30.

[58] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803. USENIX Association, November 2020.

[59] The Kubernetes Authors. Production-Grade container orchestration. https://kubernetes.io/, 2015.

[60] Timescale. Benchmarking TimescaleDB vs. InfluxDB for time-series data. https://assets.timescale.com/whitepapers/Timescale_WhitePaper_Benchmarking_Influx.pdf, 2019.

[61] Timescale. Timescale: Modern Postgres for speed, scale and AI. https://www.timescale.com/, 2025.

[62] Paul Tune and Matthew Roughan. Internet traffic matrices: A primer. http://sigcomm.org/education/ebook/SIGCOMMeBook2013v1_chapter3.pdf. Accessed: 2023-4-19.

[63] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[64] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 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*, OOPSLA 2016, pages 765–780, New York, NY, USA, October 2016. Association for Computing Machinery.

[65] Bo Yang, Huanwu Hu, Yifan Li, Yunguang Li, Xiangyu Tang, Bingchuan Tian, Gongwei Wu, Jianfeng Xu, Xumiao Zhang, Feng Chen, et al. Skynet: Analyzing alert flooding from severe network failures in large cloud infrastructures. In *ACM SIGCOMM*, pages 512–526, 2025.

[66] Yifei Yuan, Fangdan Ye, Yifan Li, Jingkai Zhang, Mengqi Liu, Yuyang Sang, Ruizhen Yang, Duncheng She, Zhiqing Ye, Tianchen Guo, et al. New evolution of Hoyan: Enhancing scalability, usability, and accuracy for Alibaba's global WAN verification. In *ACM SIGCOMM*, pages 809–825, 2025.

A Link-Invariant Imbalance at WAN *B*

Fig. 10 presents the measured link-invariant imbalance PDF at WAN *B*. First, Fig. 10(a) shows how most imbalances hold within 1%. Next, Fig. 10(b) illustrates the impact of collection times on the CDF of the imbalance. By averaging over longer periods, the imbalance decreases, trading off against a longer time before raising a potential alarm. The figure shows how averaging over 1 minute and 5 minutes yield very similar results.

B Repair Guarantees: Proof of Thm. 1

Proof. The theorem is about corrupted counters at a single link. We will distinguish two types of link:
(i) internal links, which are between two routers of the network, and where two counters can be affected; and
(ii) border links, which are between a network router and outside the network. Only the counter of the router can be affected.

First, we need to prove that the corrupted counters will not affect other links in the iterative repair computation. For instance, in Fig. 3, we want to show that a corruption of the egress and ingress counters at link $X \to Y$ will not affect the link $A \to X$. Assume that the counters at link j are corrupted. If a neighboring link j' is internal, it has 5 estimators, and j only impacts one of these, so the repair algorithm will reach the correct value for j' with confidence of at least $1-\frac{1}{5}=0.8$. Likewise, if j' is a border link, then at least two estimators out of three are unaffected and the algorithm reaches the correct value with confidence of at least $\frac{2}{3}$. Finally, if all neighboring links are unaffected, then a non-neighboring link also cannot be affected, even after several rounds of computation.

Second, we need to prove that the repair algorithm reaches the correct decision for link j. We distinguish two cases:
(i) Internal link: Both the demand-based estimator and the router-based estimators are unaffected by the corruption, since the demand vector and the neighboring link counters provide correct values. Thus, the algorithm will correctly converge with a confidence of at least $\frac{3}{5}=0.6$.
(ii) Border link: Again, the demand-based and router-based estimators are unaffected, and therefore it will reach the correct decision with a confidence of at least $\frac{2}{3}$. □

C Scaling Model: Proof of Thm. 2

Proof. Let p (respectively p') denote the probability that the invariant imbalance falls within the threshold of τ under healthy (resp. buggy) inputs, so $p > p'$. Also let Γ denote a fixed validation cutoff, set such that $p > \Gamma > p'$. Then the validation algorithm checks whether after flipping n times a coin with prob. p (resp. p') of getting heads, the proportion of heads is at least Γ. This is exactly $B_{n,p}(n \cdot \Gamma)$ (resp. $B_{n,p'}(n \cdot \Gamma)$), using the CDF $B_{n,p}$ (resp. $B_{n,p'}$) of the corresponding Binomial distribution.

As a result, FPR and $1-TPR$ have a Chernoff–Hoeffding upper bound that decreases exponentially with n [46, 52], which proves the theorem. Namely,

$$FPR \leq e^{-n \cdot D(\Gamma \| p)} \tag{5}$$

and

$$1-TPR \leq e^{-n \cdot D(\Gamma \| p')}, \tag{6}$$

where

$$D(x \| y) = x \cdot \ln\left(\frac{x}{y}\right) + (1-x) \cdot \ln\left(\frac{1-x}{1-y}\right) \tag{7}$$

is the Kullback–Leibler divergence between Bernoulli random variables with parameters x and y. □

D Repair Pseudocode

Algorithm 2 presents the pseudo-code for the repair algorithm.

E Generating Baseline Link Counters That Match Invariant Noise

We generate production-realistic telemetry counters for simulation by trying to match the link-invariant, node-invariant and path-invariant noise distributions we observe in the WAN *A* production data (Fig. 2). To do this, we first compute the expected amount of traffic on each link with the given demands and paths. Then, we perturb the link counters to account for the path-invariant noise. To do so, we add to each link some random i.i.d. noise that is distributed like the path-invariant noise distribution, and assign the noisy link value to its two counters. Next, we further modify the interface counters on either side of the link to match the three invariant noise distributions. This is done by first drawing some noise random variable x from the link-invariant noise distribution, then adding $x/2$ to one of the two counters at the end of the link, while removing $x/2$ from the other. Thus, their difference is x, following the link invariant noise, and their average has not changed, thus still obeying the path invariant noise. Finally, the router invariant noise is still unsatisfied. We similarly update counters at each router to obey it, while making sure that the distributions of the other two invariants still hold, until we converge to a satisfying result. Thus our baseline healthy bug-free telemetry will contain noise matching observed real-world data.

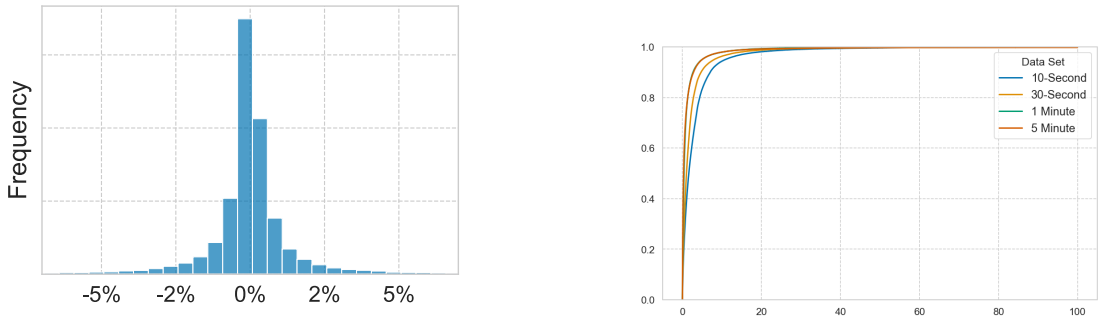
F Additional Evaluations

Do all repair components help reduce counter errors? In the GÉANT network, we assume that a router bug affects

Algorithm 2: Repair algorithm for telemetry correction.

Input: Latest measured telemetry values for each interface, stored in `telemetry_dict`,
e.g., `telemetry_dict`={`r1.eth12`: {`send_rate`: 100, `recv_rate`: 75, `status`: UP}, ... }
Static network layout information, stored in `network`,
e.g., `network`={`routers`: `[r1, r2, ...]`, `links`: `[(r1.eth12, r2.eth5), (r1.eth11, r3.eth4), ...]`}

```
locked ← {} ;                                                    // Final results, locked values
while size(locked) < num_counters do
    possible_values ← {};
    foreach link_id ∈ network.links do
        if link_id ∈ locked then
            possible_values[link_id] ← [locked[link_id].value];
            continue;
        src_intf, dst_intf = link_id[0], link_id[1];
        if src_intf is internal then
            src_measurement ← telemetry_dict[src_intf][send_rate];
            possible_values[link_id].append(src_measurement);
        if dst_intf is internal then
            dst_measurement ← telemetry_dict[dst_intf][recv_rate];
            possible_values[link_id].append(dst_measurement);
        predicted_val ← get_predicted_traffic(paths, demand, link_id);
        possible_values[link_id].append(predicted_val);
    votes ← {} ;                                  // (src_intf, dst_intf): [(v1, weight1), ...]
    // Populate votes from router beliefs.
    foreach router ∈ network.routers do
        local_links ← get_links_attached_to(router) ;        // Any links with src or dst intf at this router
        rtr_predicted_vals ← {} ;
        for i ← 1 to N do
            assignment ← {};
            foreach link_id ∈ local_links do
                assignment[link_id] ← random.choice(possible_values[link_id]) ;
            foreach link_id ∈ local_links do
                rtr_predicted_vals[link_id].append((∑ assignments) - assignment[link_id]);
        foreach link_id ∈ local_links do
            val, conf ← most_frequent_value_with_confidence(rtr_predicted_vals[link_id]) ;
            votes[link_id].append((val, conf));
    // Populate votes from possible_values
    foreach link_id, vals_list ∈ possible_values do
        foreach val ∈ vals_list do
            votes[link_id].append((val, 1.0));
    assignment_scores ← {} ;                        // Lock the value with the highest majority confidence
    foreach link_id, votes_list ∈ votes do
        if src_intf ∈ locked then
            continue;
        cumulative_scores ← group and sum votes_list by value with ≤ 0.03% threshold;
        best_val, score ← max(cumulative_scores);
        assignment_scores[link_id] ← (best_val, score);
    link_id, (best_val, score) ← max(assignment_scores);
    locked[link_id] ← (best_val, score);
return locked
```



(a) Link invariant over 30 second window.

(b) Impact of collection time

Figure 10: Measured imbalance for the link invariants at a large production WAN *B*.

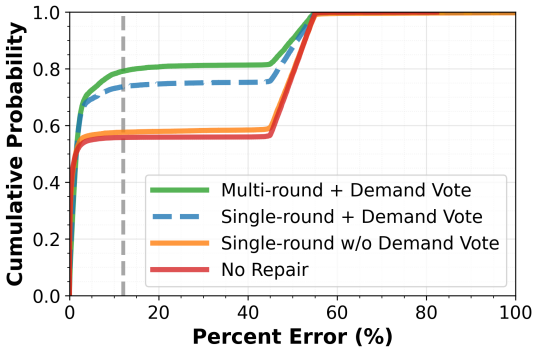


Figure 11: Impact of the repair components on the CDF of the counter error in GÉANT.

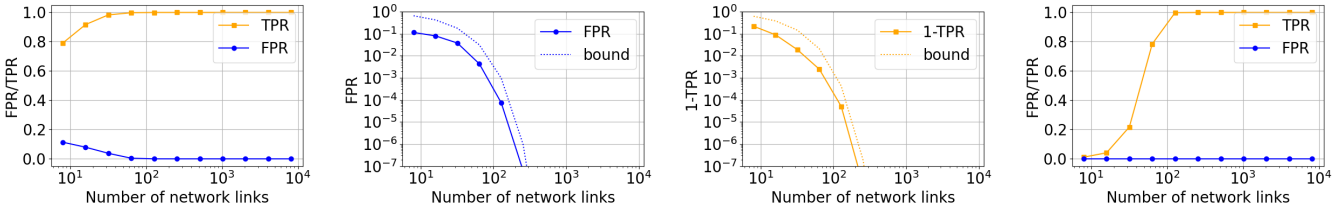
45% of the counters, and they are scaled down by a random factor chosen uniformly at random in the range [45%,55%]. Fig. 11 illustrates the contribution of the various components of the repair algorithm on the CDF of the counter error – we see for the no repair baseline (red solid line), 55% of counters have only noise, and the remaining 45% have errors between 45% and 55%. If we run a single round of the repair algorithm without the fifth vote of the demand-induced link estimate, we only correct an additional 3-4% of counters. With a single round that includes all five votes, it goes up to 75% of counters with error under 10%. Thus, this fifth tie-breaking vote brings the most significant contribution. Finally, if we use the full repair with all gossip rounds, it increases to over 80% of counters having less than 10% error, *i.e.,* the CrossCheck repair corrected about 2/3 of the router bug-induced errors (from 45% to 15%).

Does CrossCheck improve exponentially in larger networks? Fig. 12 plots the FPR and TPR under the scaling model of Thm. 2, where the path invariant imbalance distributions are assumed to be the same for all links in all networks. Specifically, for healthy inputs, the model assumes the path invariant imbalance distribution that was measured in production WAN *A*. For buggy inputs, it adds a Gaussian-distributed imbalance to this distribution using $\mathcal{N}(5,5)$. In Fig. 12(a), given a fixed validation cutoff ($\Gamma = 0.6$), both TPR and FPR quickly converge to 1 for large networks, but are not optimal for small ones. Figs. 12(b) and 12(c) provide a close-up view of FPR and $1-TPR$, illustrating how they decrease exponentially fast with the number of links together with their upper bound, as expected from Thm. 2 and its proof (note that the x axis also uses a log scale).

Fig. 12(d) sets a different cutoff for each network size, so that $FPR \leq 10^{-6}$, corresponding to at most one false alarm every ten years assuming a validation every five minutes. Since there is a TPR vs. FPR tradeoff, TPR becomes worse, and suffers more for small networks. Given that our considered WANs have at least 54 uni-directional links (for Abilene), and modern real-world WANs are much larger, this indicates that CrossCheck should be quite efficient.

G Theoretical Aspects of Input Validation

Guessing inputs. The path invariant relates input demands to router counters. It is natural to ask whether we could guess input demands given low-level network telemetry. Compressed sensing (CS) and low-complexity iterative message-passing algorithms such as belief propagation (BP) are efficient at inferring the values of a set of random variables, given a set of constraints that induce relations between them using a sparse graph structure, *i.e.,* such that each constraint depends on a small number of variables on average [45, 13, 16]. For instance, Counter Braids [41, 42] iteratively provides upper and lower bounds on the estimated random variables. These bounds converge to a solution given a sufficient number of constraints. Furthermore, matrix completion approaches [30, 31, 9] attempt to recover a matrix from a sampling of its entries. However, our case differs meaningfully. First, the invariants do not suffice to reconstruct the demand matrix, as shown in the counter-example below. Second, we do not know what entries in our corrupted input have been corrupted, and we do not know how they have been corrupted, a model not dealt with in these frameworks. In our case, the bounds provided by the Counter Braids are too wide and miss an overwhelming majority of the data corruption in most corruption scenarios.



(a) Fixed cutoff **(b)** Zoom on FPR **(c)** Zoom on $1-TPR$ **(d)** Variable cutoff

Figure 12: FPR/TPR scaling model as a function of the number of network links. (a) Fixed validation cutoff, with exponentially decreasing (b) FPR and (c) $1-TPR$. (d) Tuning the cutoff for each network size to attain a near-zero FPR.

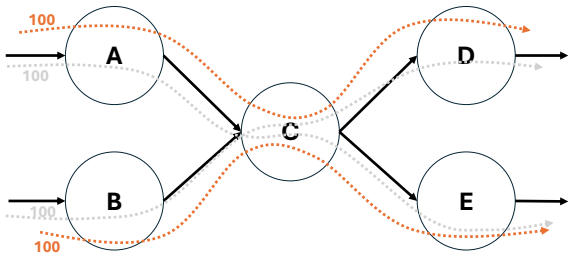


Figure 13: Detection counter-example.

Counter-example to guessing demands. The path invariant asserts that after taking into account the routing, the total demand for a link should equal the router link counters (Eq. (4)). It is natural to ask whether we could simply reverse-engineer, and guess the input demands from the low-level network telemetry. This would be a simple way to check the input demands.

Fig. 13 provides a counter-example. The demand includes two (source, destination) flows: (A,D) and (B,E). Each flow is of size 100. All of the link counter values will also be 100. However, if a bug reports the demand as consisting of the two pairs (A,E) and (B,D), nothing would change in the link counter values. Thus, both the healthy and buggy demands would yield the same telemetry signals. This illustrates why we cannot unequivocally establish the demand from the lower-level telemetry.