nsdi '26

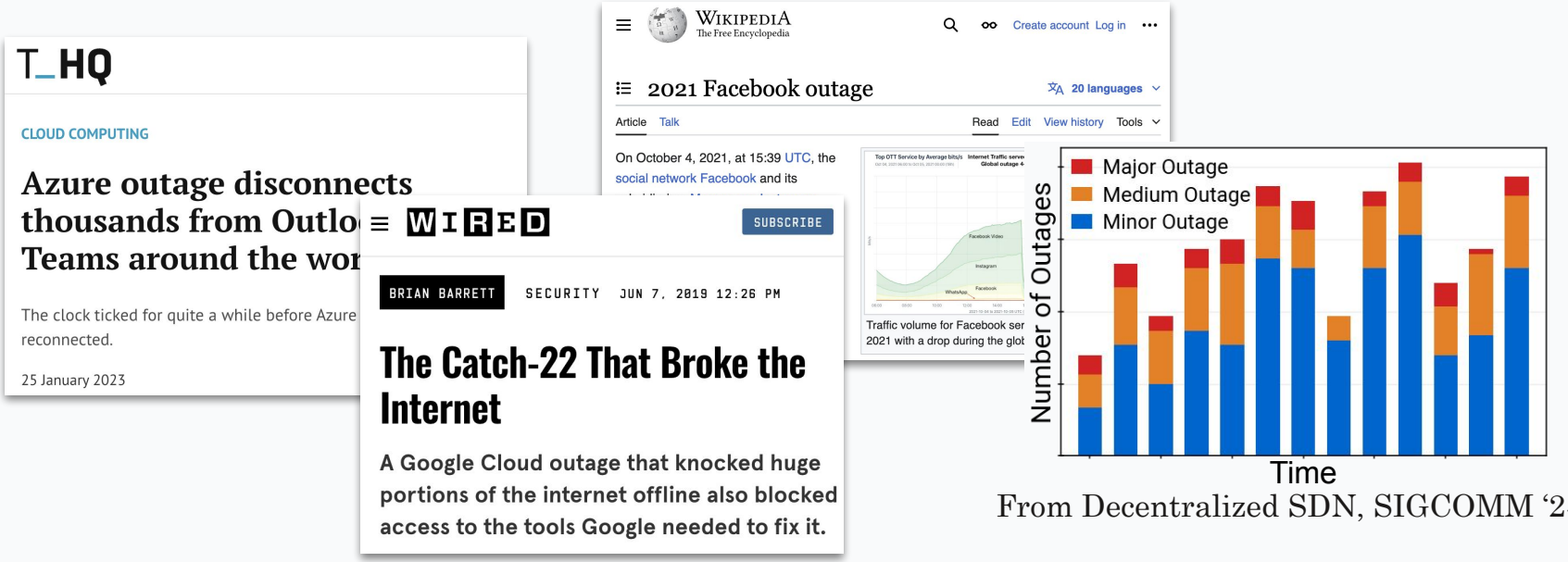
CrossCheck: Input Validation for WAN Control Systems

Alexander Krentsel[12], Rishabh Iyer[1], Isaac Keslassy[3], Bharath Mohdipalli[2], Sylvia Ratnasamy[12], Anees Shaikh[2], Rob Shakir[2]

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

Network outages continue to happen

Despite the best efforts of our community…



From Decentralized SDN, SIGCOMM ‘24

What are we missing?

Important root cause class: incorrect control inputs

Conducted analysis of high-impact SDN WAN outages over 5 years…

⇒ Over 1/3rd root-caused to <u>incorrect inputs</u>.

Many existing approaches *cannot help*:

- Verification:
  - Batfish [NSDI ‘15], VeriFlow [NSDI ‘13]
  - Header Space Analysis [NSDI ‘12]
- Simulation/Emulation
  - SimBricks [SIGCOMM ’22]
  - CrystalNet [SOSP '17]
  - Mininet [HotNets ‘10]
- Testing:
  - NetCastle [NSDI ‘24]
  - Ixia (Keysight)

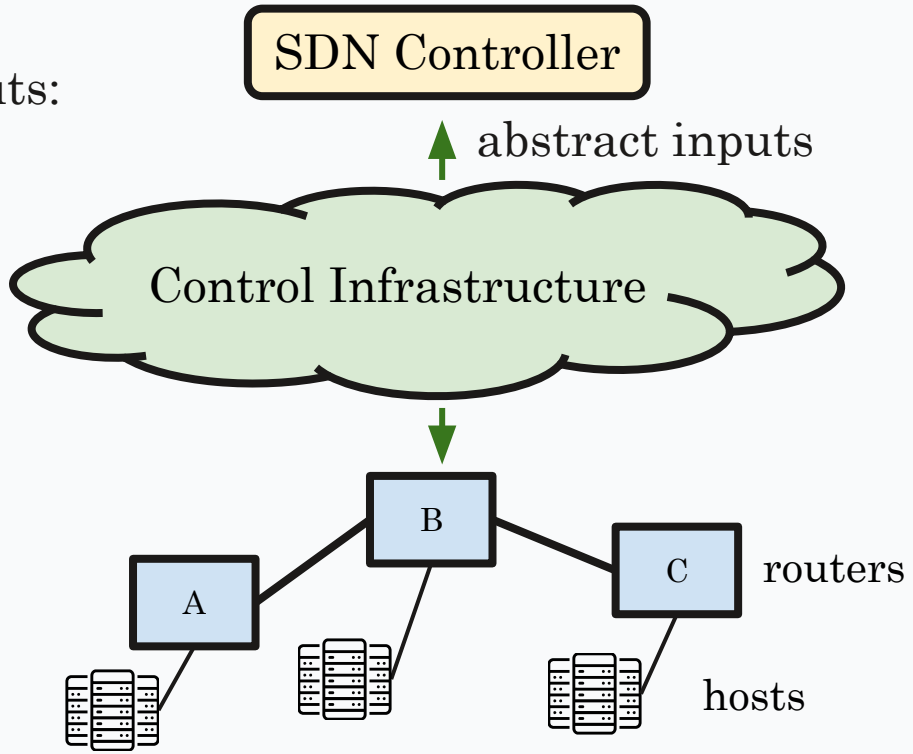
Existing techniques focus on dataplane or controller correctness, not the input being fed to it.

What inputs?

SDN controller makes decisions for the network, taking two main (abstract) inputs:

(1) Topology

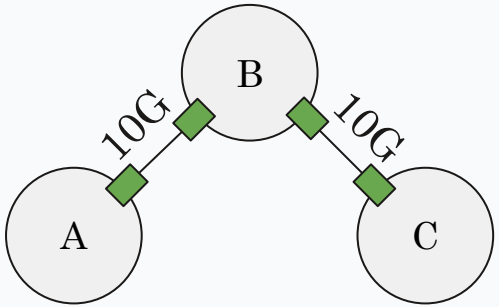
(2) Demand

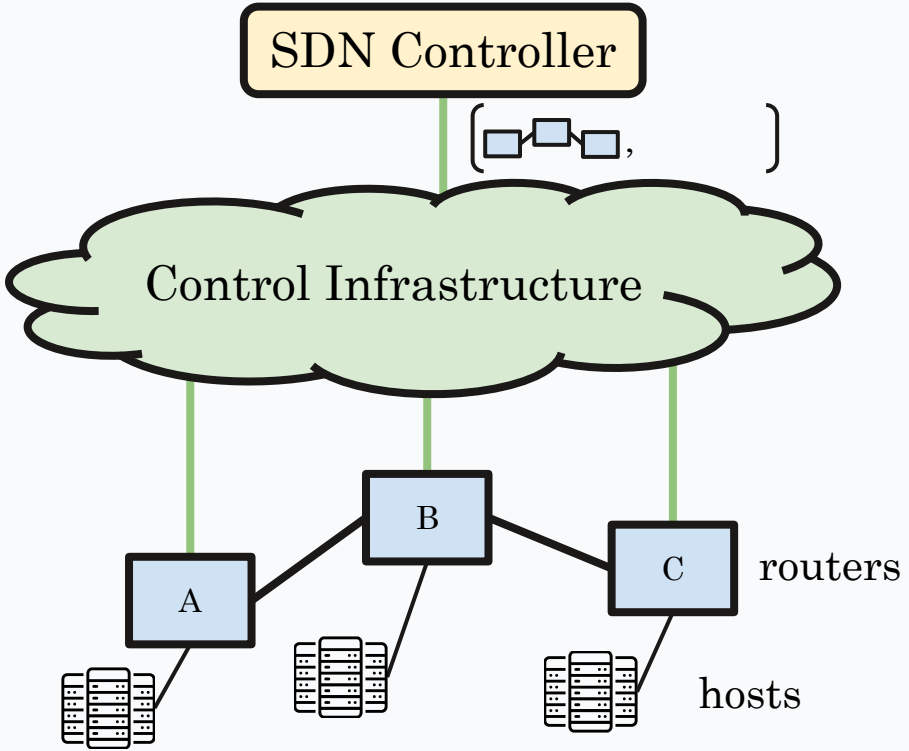


What inputs? Topology

Topology:

- Constructed from router-reported interface status & capacities
  - Collected from router telemetry
  - Aggregated across interfaces
- Abstract network graph

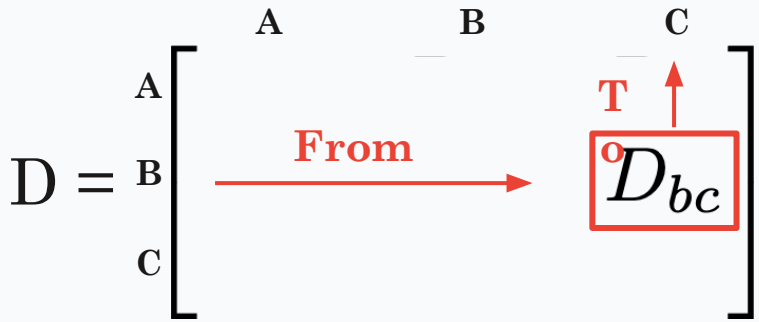


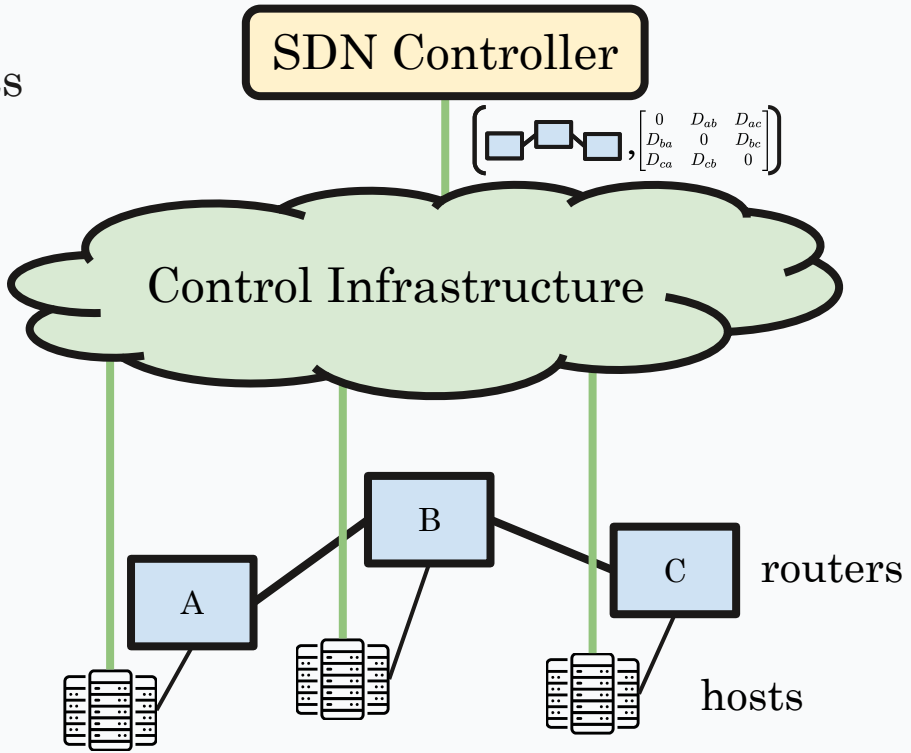


What inputs? Demand

Demand:

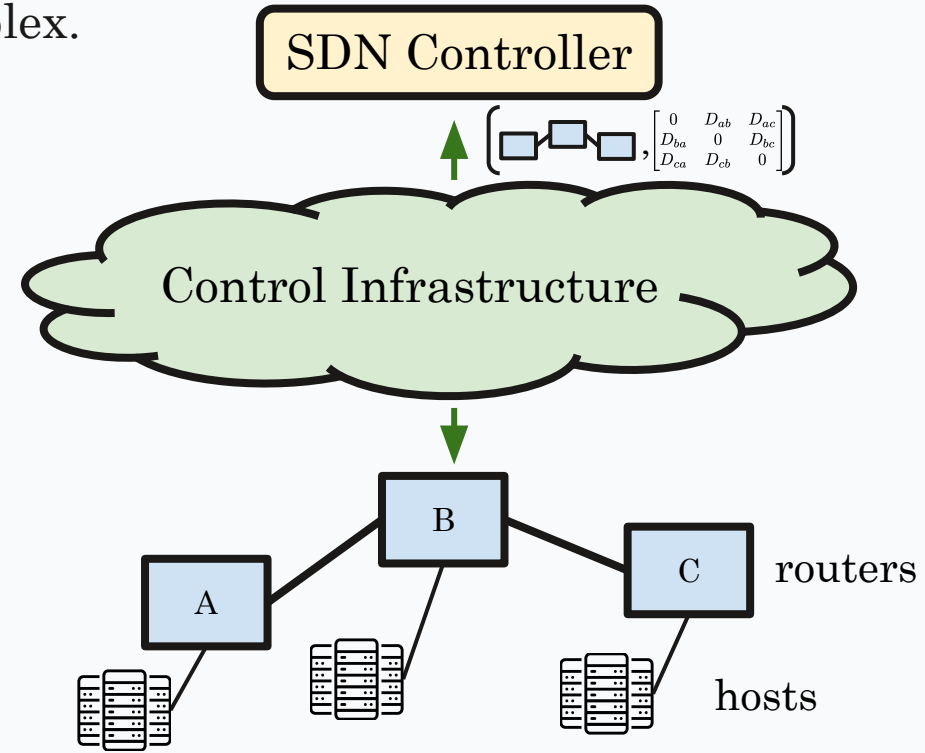
- Constructed from host-measurements
  - Collected from host servers
  - Aggregated across jobs
- Rate of traffic sent onto the network, destined for egress.





Reality: input is not simple

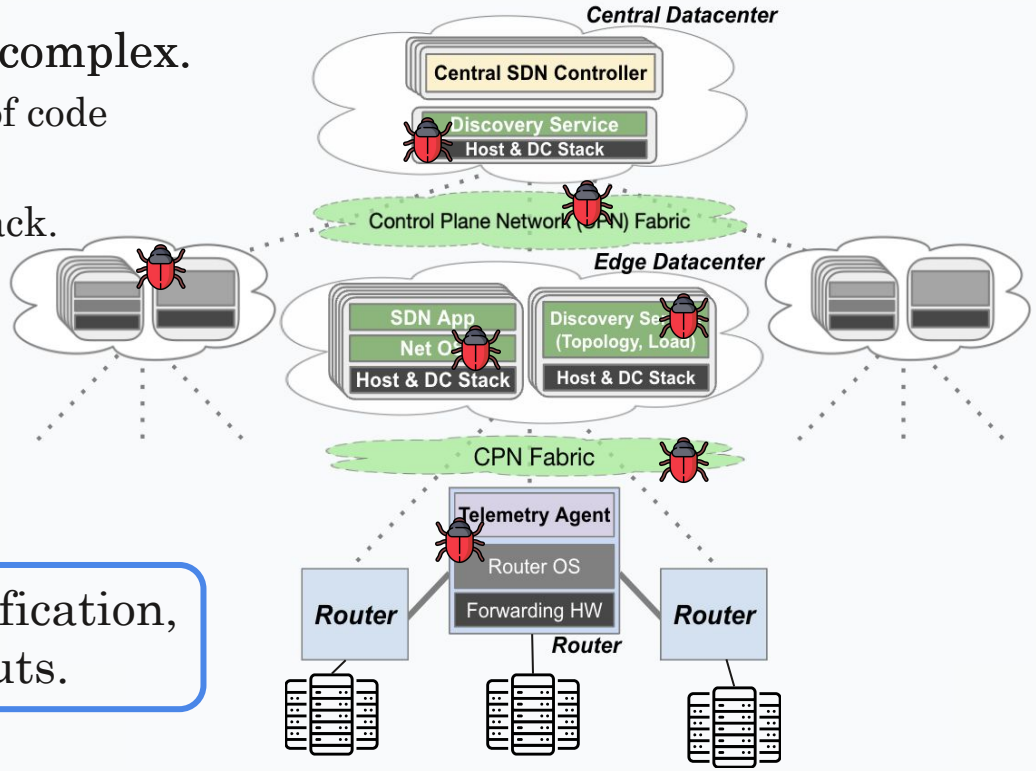
Conceptually simple, operationally complex.



Reality: input is not simple

Conceptually simple, operationally complex.

- Control infra spans millions of lines of code across a half-dozen distinct systems.
- Bugs can happen anywhere in the stack.



No amount of controller testing, verification, etc. can help with incorrect inputs.

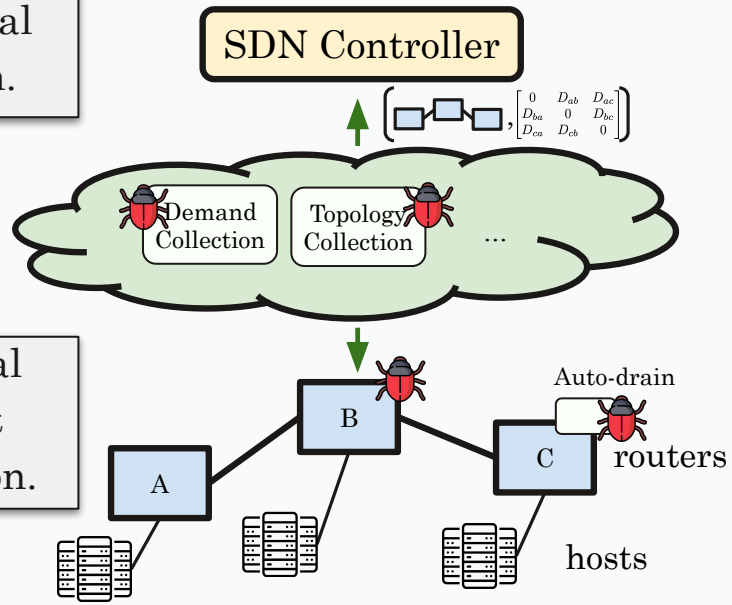
Incorrect inputs can have catastrophic consequences

Outage 1. Bug in demand instrumentation service led to demand collected incorrectly, resulting in partial demand fed to SDN controller. Severe congestion.

Outage 2. Bug causes topology instrumentation service to aggregate topology before all routers report, feeding partial topology to controller.

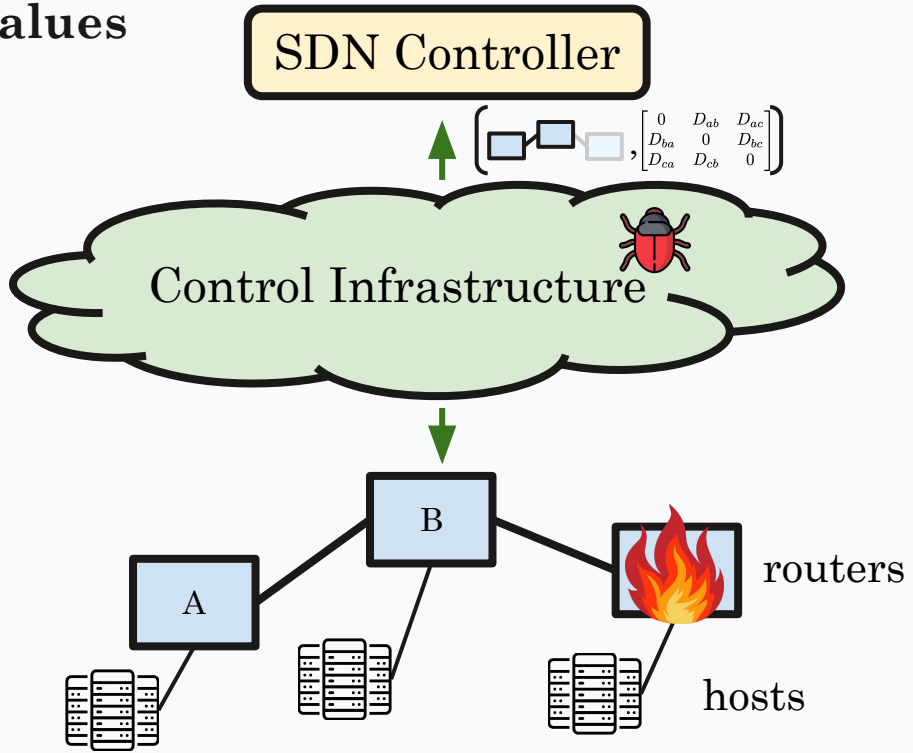
Outage 3. Overly-sensitive automatic drain signal gets triggered, incorrectly draining a significant number of perfectly-fine routers. Severe congestion.

Outage 4. Bug in vendor OS causes telemetry messages to oscillate between 0 and actual. Triggers flap protection drains.



So, just check your inputs?

Incorrect inputs are often possible values but *not current ground truth*.



No way to distinguish from input alone

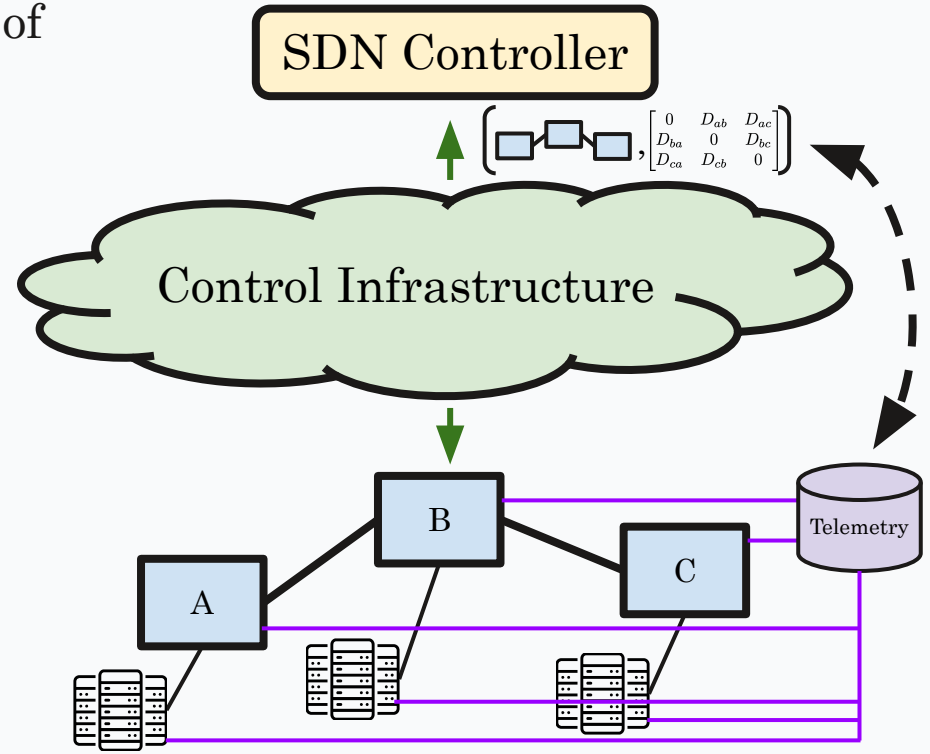
This means that anomaly detection, edge case testing, etc. cannot find them.

Insight: observe multiple indicators of ground truth

Many additional independent indicators of ground truth, collected for obserability:

- Interface byte counters
- Packet drop counters
- Forwarding entries
- Bidirectional Forwarding Detection (BFD) link monitoring updates
- Probes

Ought to agree because they reflect same current ground truth…



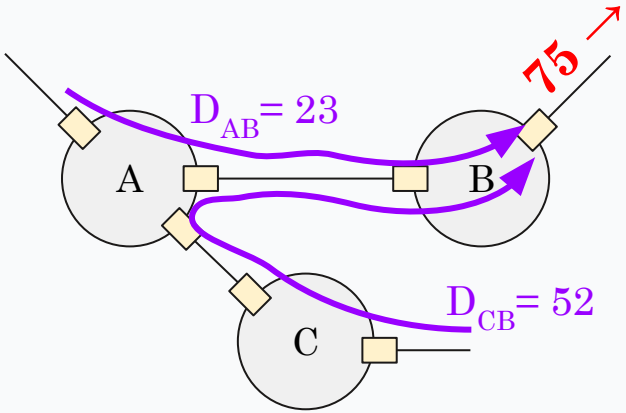
Key Idea #1: *Cross check* inputs and telemetry

Certain ***invariants*** hold between inputs and telemetry…

$$D = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 23 & 55 \\ 18 & 0 & 41 \\ 30 & 52 & 0 \end{bmatrix} \end{array}$$

Example demand invariant:

∑ demands to node 〓 egress interface counter



If invariant does not hold, something is wrong.

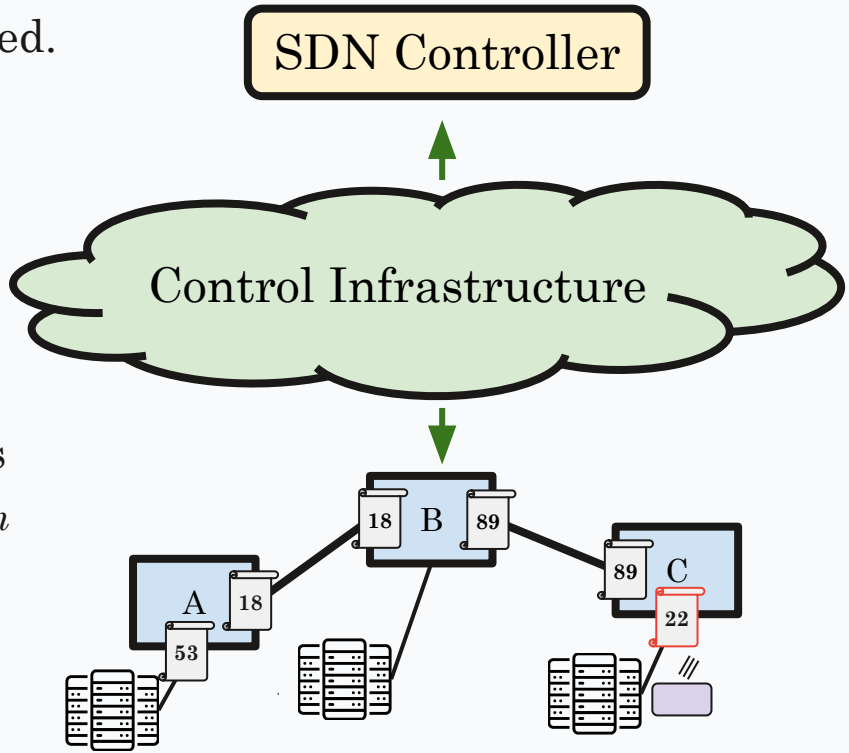
Challenge: solving the blame game

When we detect telemetry and inputs disagree, which is wrong?

A path forward available through the *redundancy* available in telemetry…

Key idea #2: Redundancy in telemetry enables corroboration

Network signals are naturally interconnected.



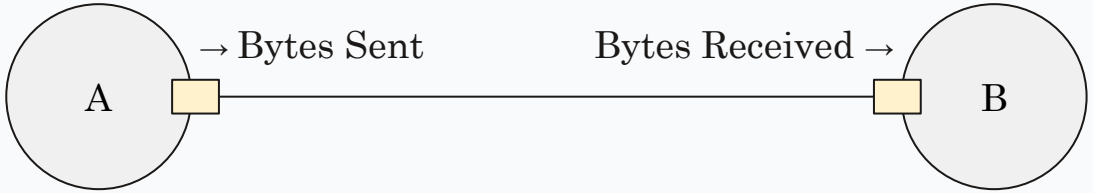
Actions reflected in multiple measurements

→ provides *redundancy* and thus *corroboration*

Redundancy enables detection

Many examples of redundancy in network data…

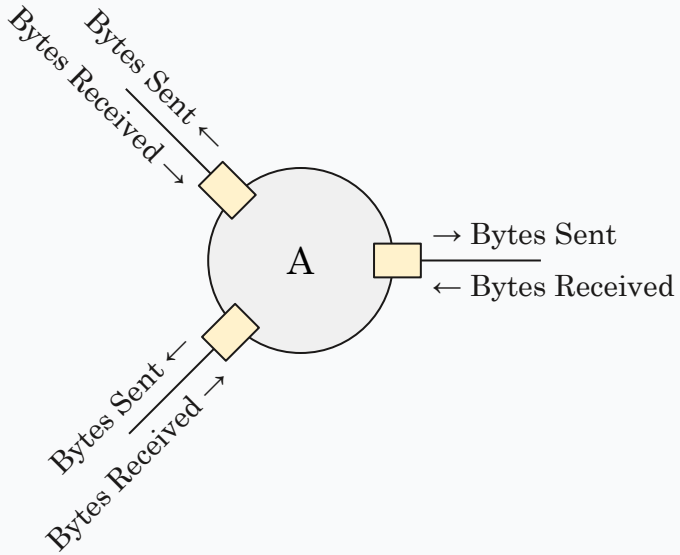
- **R1. Symmetry** across two ends of link: bytes in = bytes out.



Redundancy enables detection

Many examples of redundancy in network data…

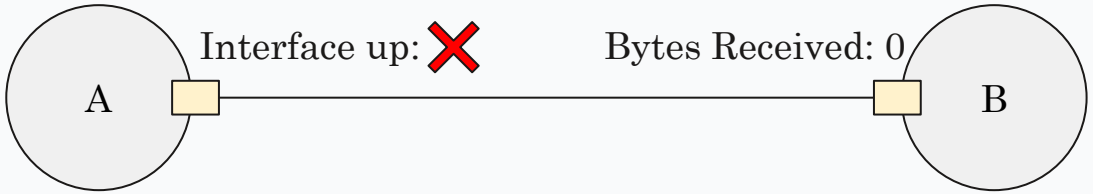
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).



Redundancy enables detection

Many examples of redundancy in network data…

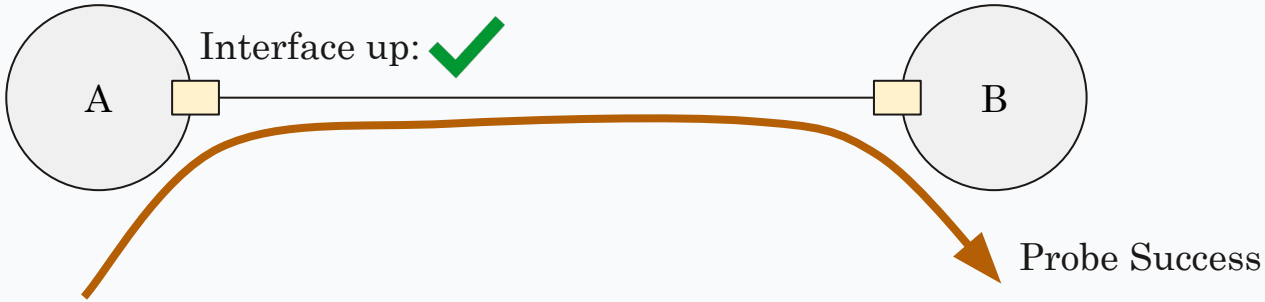
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.



Redundancy enables detection

Many examples of redundancy in network data…

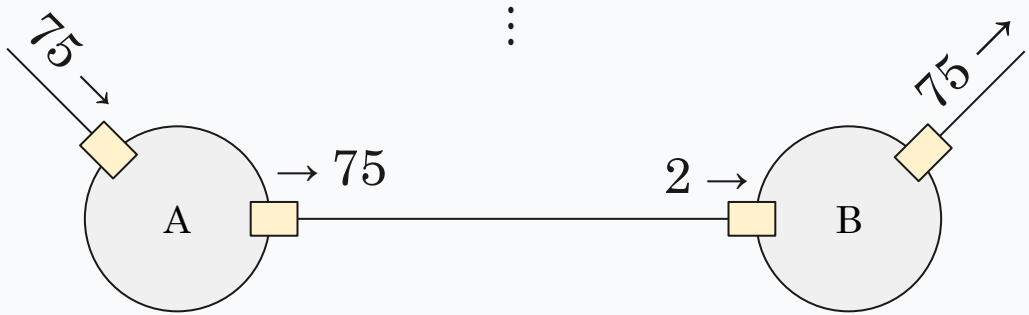
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.
- **R4. Manufactured signals**: probe confirms link up/down.



Redundancy enables detection *and repair*

Many examples of redundancy in network data…

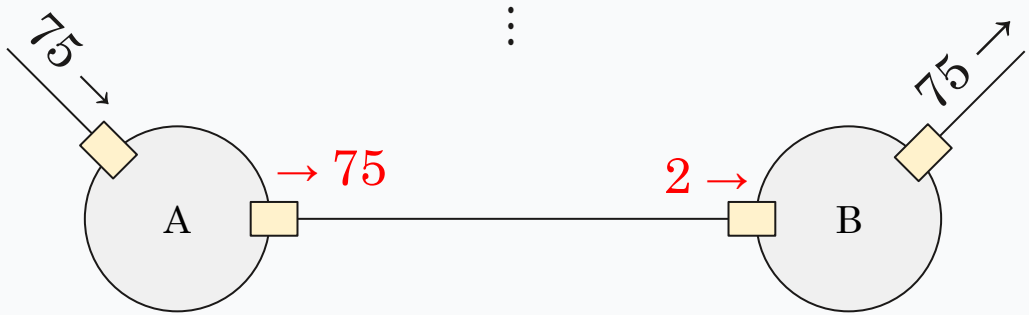
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.
- **R4. Manufactured signals**: probe confirms link up/down.



Redundancy enables detection *and repair*

Many examples of redundancy in network data…

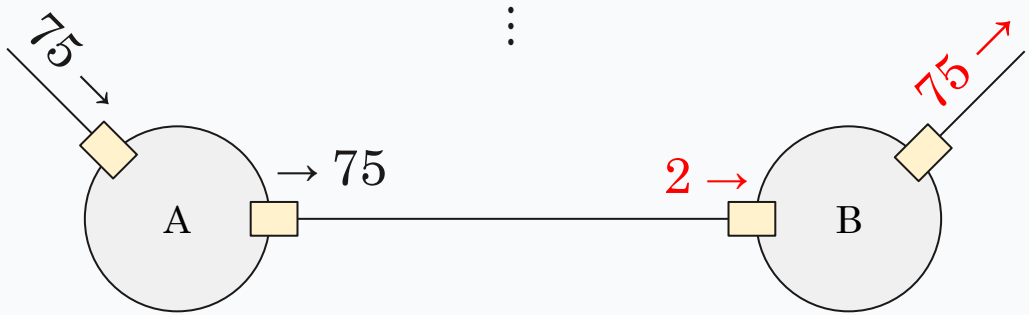
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.
- **R4. Manufactured signals**: probe confirms link up/down.



Redundancy enables detection *and repair*

Many examples of redundancy in network data…

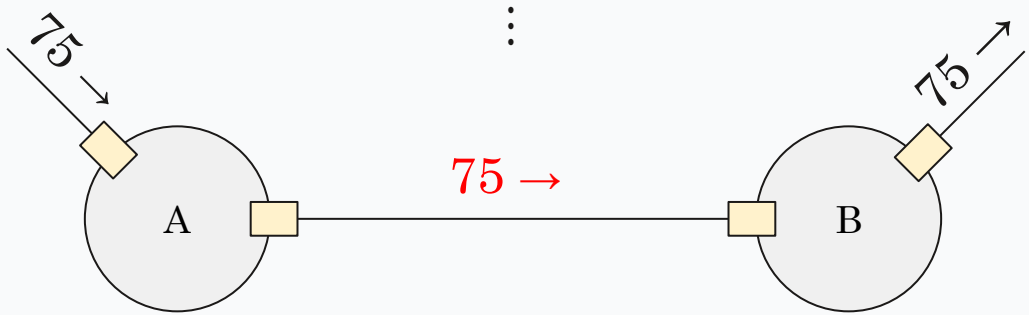
- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.
- **R4. Manufactured signals**: probe confirms link up/down.



Redundancy enables detection *and repair*

Many examples of redundancy in network data…

- **R1. Symmetry** across two ends of link: bytes in = bytes out.
- **R2. Flow conservation** across router ports: sum in = sum out (+ drops).
- **R3. Alternative signals**: link down ⇒ intf count = 0.
- **R4. Manufactured signals**: probe confirms link up/down.

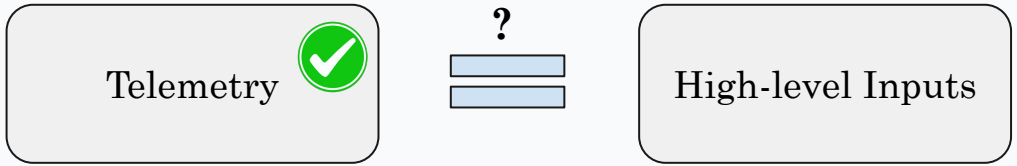


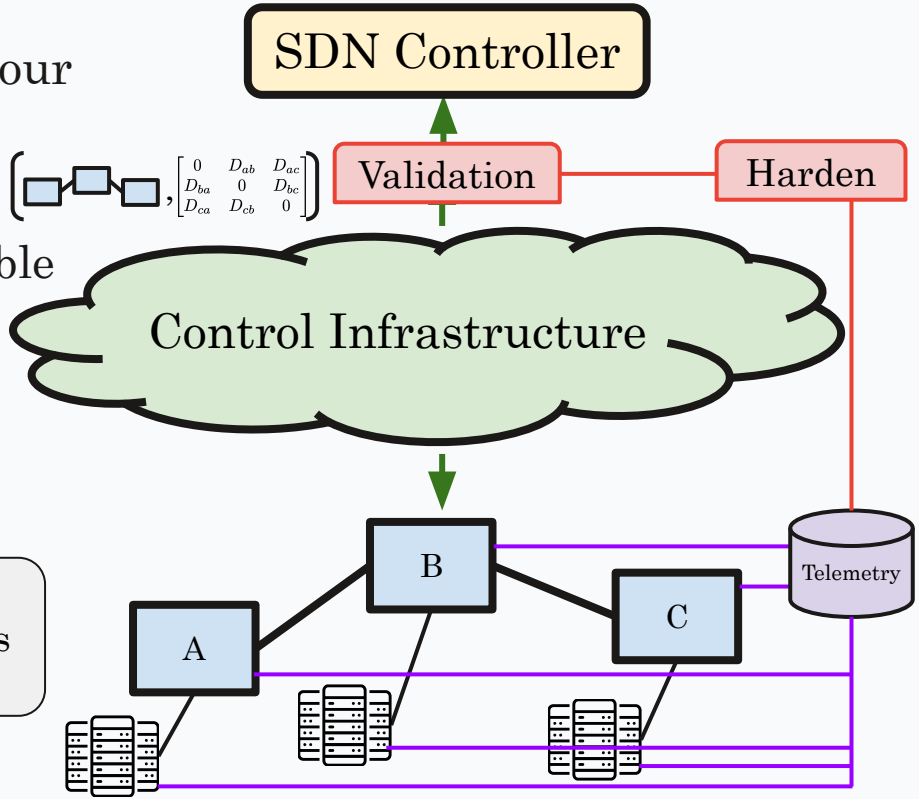
Our system: CrossCheck

We combine these two insights to design our system, CrossCheck, which…

(1) "**Harden**": Detects and repairs available telemetry via redundancy

(2) "**Validate**": Cross-checks inputs with hardened telemetry



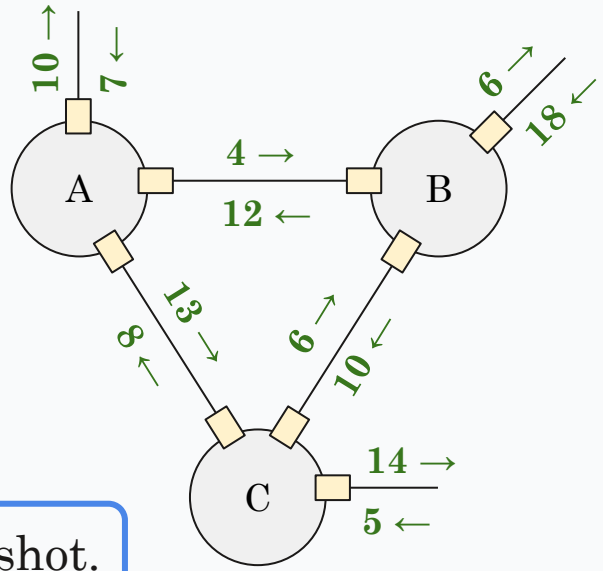


Harden: produce self-consistent telemetry snapshot

We introduce an iterative algorithm for telemetry repair.

Three steps per iteration:

1. Score each value's self-consistency
2. Select the highest-consistency value
3. "Harden" that value, and repeat



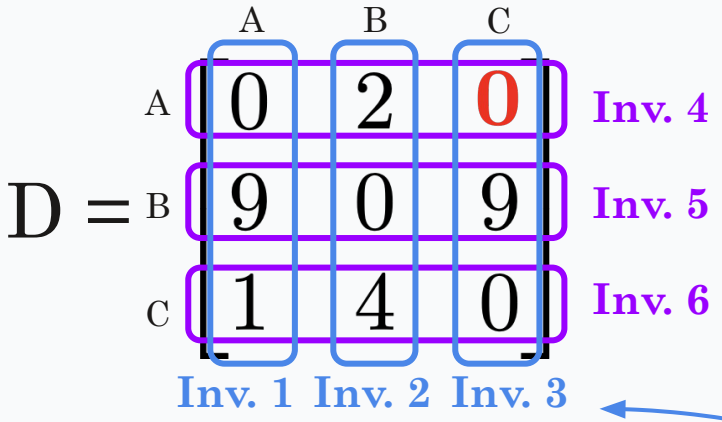
Produces a repaired and trustworthy telemetry snapshot.

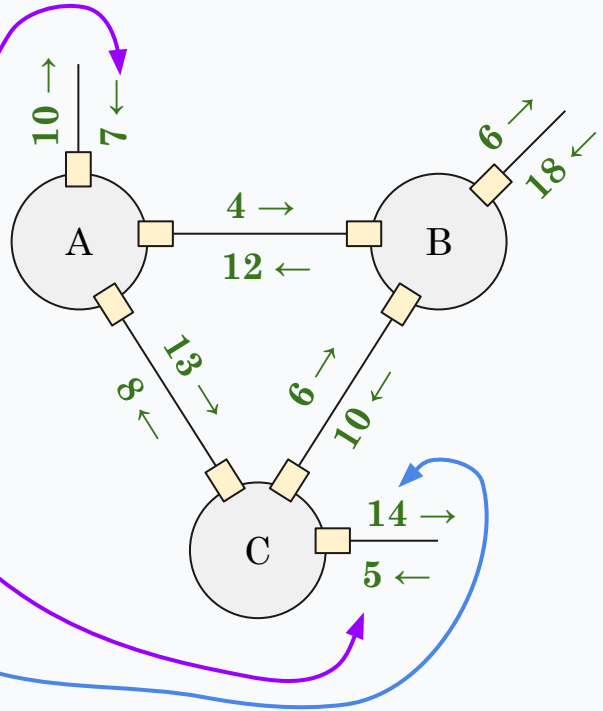
 (see full details on algorithm in the paper: Democratic Trust Propagation (DTP))

Validate: invariant checking

Check all invariants across low-level telemetry and abstract inputs

- < **Γ** % of invariants pass, declare input *buggy*





Evaluation: does CrossCheck work?

Goal: catch as many incorrect inputs, *without false positives*.

Core questions:

(1) how sensitive to buggy inputs? (maximize True Positive rate)
(2) how resilient to buggy telemetry? (minimize False Positive rate)
(3) is it effective in the real world?

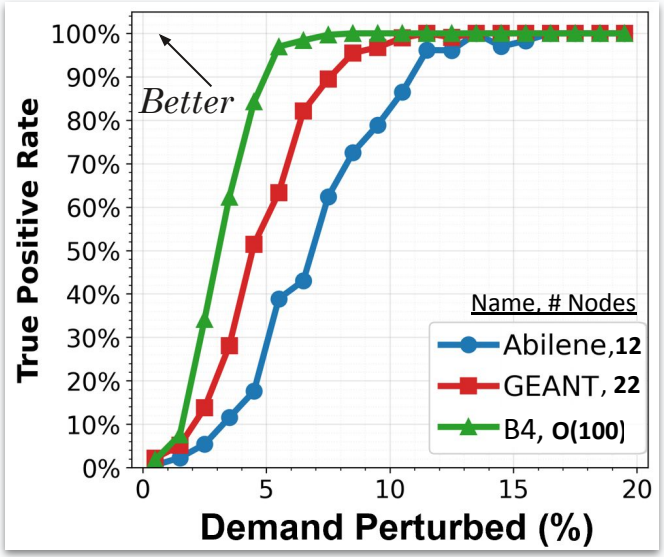
Implemented CrossCheck and tested it via:

- Shadow deployment in Google's WAN
- Real-world traces (Google, Abilene, GEANT) with synthetically injected bugs (in input, telemetry, paths)

Does CrossCheck catch buggy inputs?

Buggy demands (true positives):

- Telemetry & demand containing regular production noise distributions.
- Demand entries made buggy by scaling up or down at random by varying degree.

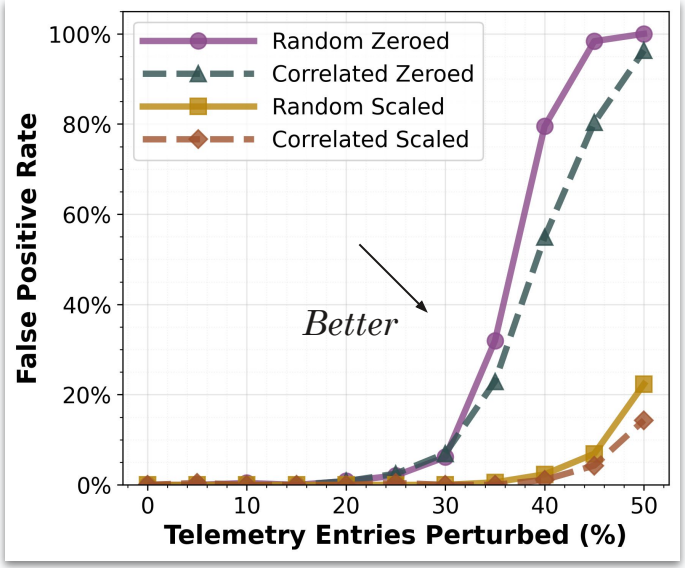


CrossCheck is *very* effective, detecting nearly 100% of demand perturbations over 5% for B4, and *improves* with network size.

Does CrossCheck trigger falsely on buggy telemetry?

Buggy telemetry, regular demand:

- Telemetry & demand containing regular production noise distributions.
- Telemetry perturbed at random by scaling up or down or zeroing
  - Uniform random: corresponds to drops
  - Correlated: corresponds to router bugs

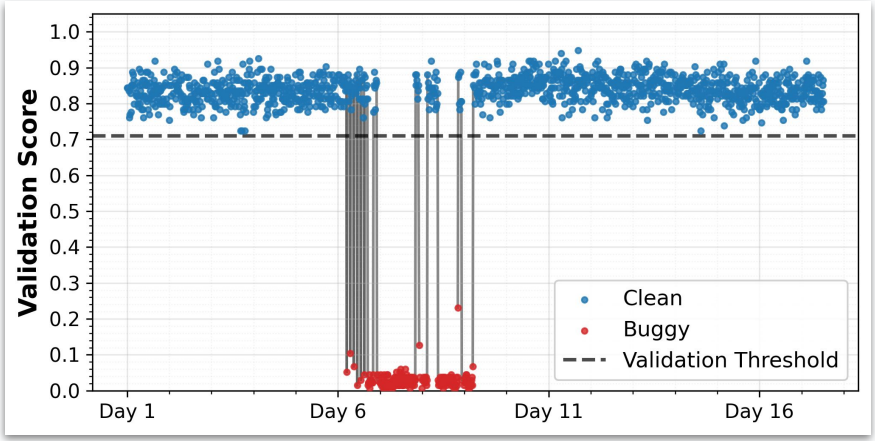


CrossCheck is resilient to false positives for up to ~25% of telemetry perturbed, because *repair* is good.

Does CrossCheck work in the real world?

Ran 1-month shadow deployment…

- Reading live production shadow datastore
  - Real demand, telemetry, and paths
- Observed *zero* false-positives
- Immediately detected buggy rollout affecting shadow datastore
  - shadow datastore used for planning



CrossChecks *correctly* and *immediately* detected a real-world scenario of corrupted inputs.

Further details

Paper link

See paper for discussion on additional questions…

- Does CrossCheck work for other types of inputs?
- Is CrossCheck resilient to bugs in other low-level signals?
- Is there a principled mechanism to set CrossCheck's hyperparameters?
- Do CrossCheck's principles generalize to other control systems?

+ further eval on repair quality, proofs of repair success

We believe CrossCheck can apply to a wide range of control systems where redundancy is available.

Questions?

Alexander Krentsel – akrentsel@berkeley.edu