FAst In-Network *GraY* Failure Detection for ISPs

Edgar Costa Molero
ETH Zurich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@ucl.ac.uk

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

ABSTRACT

Avoiding packet loss is crucial for ISPs. Unfortunately, malfunctioning hardware at ISPs can cause long-lasting packet drops, also known as gray failures, which are undetectable by existing monitoring tools.

In this paper, we describe the design and implementation of FANcY, an ISP-targeted system that detects and localizes gray failures quickly and accurately. FANcY complements previous monitoring approaches, which are mainly tailored for low-delay networks such as data center networks and do not work at ISP scale. We experimentally confirm FANcY's capability to accurately detect gray failures in seconds, as long as only tiny fractions of traffic experience losses. We also implement FANcY in an Intel Tofino switch, demonstrating how it enables fine-grained fast rerouting.

CCS CONCEPTS

• **Networks** → **Network measurement**; Network simulations; **Programmable networks**; *In-network processing*;

KEYWORDS

Failure detection, Measurements, Network Hardware, Programmable data planes

ACM Reference Format:
Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2022. FAst In-Network *GraY* Failure Detection for ISPs. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544216.3544242

1 INTRODUCTION

Avoiding packet loss is so critical to ISPs that research and industry efforts focus increasingly on ensuring minimal downtime upon failures (e.g., [26, 32, 39]). A major result of past efforts is that hard failures affecting all packets crossing a link or node are typically detected and dealt with quickly thanks to the BFD protocol [28].

In practice, however, malfunctioning hardware often causes packet losses only for subsets of packets sent over a link. Table 1 (further discussed in §2.2) shows representative examples of device bugs in this category. Additional examples include misplaced line cards and bent or dirty fibers [46].

In this paper, we call *gray failure* any hardware malfunction that causes non-transient packet loss on a subset of the traffic forwarded by any packet-forwarding device – which we generally call a switch. Consistent with our definition, we do not classify congestion as a gray failure.

As confirmed by a survey we conducted (see §2.1), ISP operators consider gray failures a major concern, and lack techniques to detect and locate them. Indeed, they often become aware of gray failures only when customers complain about the failure-induced packet loss, which they end up troubleshooting for days or weeks.

The reason why existing techniques are ineffective is that detecting and localizing gray failures requires analyzing *all* the traffic. Hello protocols such as BFD do not work because most gray failures do not impact messages from these protocols. ISP monitoring tools, such as NetFlow [22] or sFlow [36], do not help either: since they rely on random packet sampling for scalability, they are unable to support fine-grained traffic analyses (as also shown in [41]), which would be needed to spot gray failures. Finally, mechanisms internal to switches, such as deflection on drop [44], do not capture several failure cases, including those where the drop flag is not correctly set on packets because of memory corruption, and link-level failures.

Of course, gray failures are not specific to ISPs, and recent contributions have proposed gray failure detectors for data center and cloud networks. Those detectors' designs, however, do not match the peculiarities of ISP networks: they either require control of end hosts [17, 23, 37, 40], assume low packet loss rates and extremely high-speed interfaces between the control and data planes (e.g., [31]), or require limited links delay and traffic volumes (e.g., [44]). We review the unsuitability of recent related work and simple system designs in §2.

Vision. We aim at designing an accurate and fast gray failure detector for ISPs. Similar to BFD, the immediate application of such a detector would be to support *selective fast rerouting on gray failures* – i.e., rerouting traffic only for the disrupted traffic, as fast as possible. We also envision that in the future, a gray failure detector may assist operators in finding the root cause of gray failures, and enable new control- and data-plane applications, such as automated failure repair through ad-hoc forward error correction mechanisms or hardware reconfiguration (e.g., [43]).

Problem statement. We focus on the following question:

> *Can we build an ISP-targeted system able to detect and localize intra-domain gray failures in seconds?*

By localizing we mean identifying both the switch port suffering from a gray failure and the affected traffic. Note that our problem statement does not directly target root cause analysis, nor automated remedies to the detected failures.

FANcY. We present FANcY, a gray failure detector tailored to ISPs. FANcY relies on an inter-switch protocol enabling data planes to synchronize packet counters and detect packet losses by comparing the values of those counters. Counters provide the minimal information needed to localize gray failures; frequently exchanging them provides detection speed and scalability (e.g., consumed memory).

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544242

Real examples of ***unwanted*** traffic drops affecting…	…***some*** packets	…***all*** packets
…***one*** or ***some*** IP prefixes	Neighbor Solicitation [10] or BGP [9] packets	Packets sent from a specific line card [1] Specific IP prefixes [3]
…***all*** IP prefixes	With specific sizes [2] With IP ID field `0xE000` [4] With wrong CRC [7, 8]	Traffic on certain ports [6, 12] Interface flaps [11, 13]

Table 1: Representative examples of gray failures plaguing major routing devices (from Cisco and Juniper bug reports).

Figure 1 shows FANcY's interface, and its role within our envisioned in-network reaction approach. As input, FANcY takes the specification of which entries the operator or the applications using FANcY are interested in monitoring, and the memory budget per switch. Every time FANcY detects packet drops induced by a gray failure, it flags the entries and ports affected by the failure.

In FANcY, an *entry* indicates a subset of the header space defined by a match rule on packets. For example, Figure 1 shows that operators can specify destination prefixes as entries, which would be reasonable if they aim to support selective fast rerouting in destination-based routed networks. We however remark that future applications can dynamically define the entries monitored by FANcY, for example, for root cause analyses – e.g., to assess losses per packet size or per value of specific IP fields.

Since fundamental limits constrain how many entries can be monitored with the limited memory available on switches, FANcY offers two levels of accuracy for entries to be monitored: high priority, and best effort. Each high priority entry is tracked with a *dedicated counter*. Best effort entries are collectively monitored with a *hash-based tree*. Contrary to existing sketches, the hash-based tree stores aggregated counters without compressing information, and is explored *in hardware*, at runtime, to identify faulty entries, in-switch and at line rate. FANcY's interface, for example, allows operators to monitor all destination prefixes, while also maximizing accuracy and reactivity for the ones driving most Internet traffic, which are typically few [38]; we assume this to be a common goal for ISP operators. If operators want to monitor a more limited set of entries, they can also specify all entries as high priority. The system returns an error, if the set of high-priority entries cannot be supported with the memory budget specified in input.

We implement FANcY in ns-3 [16], and evaluate its ability to capture gray failures through extensive simulations. Since FANcY is traffic-driven, we first assess the minimal traffic requirements that allow it to quickly and accurately localize failures. Our experiments indicate that those requirements are quite minimal for ISPs' traffic. We then experiment with real traffic traces, and confirm FANcY's potential to work well in real ISPs. Results indeed show that FANcY is able to detect and localize gray failures unless they cause loss of tiny amounts of traffic: as such, they confirm that FANcY would protect the vast majority of ISPs' traffic.

We also implement a prototype of FANcY in P4, and deploy it to an Intel Tofino switch. We use this implementation to demonstrate that FANcY enables sub-second selective fast rerouting. Combined together, FANcY and the rerouting application built on top of it take significantly fewer resources than reference P4 applications such as switch.p4. The source code of our FANcY implementations is available at https://github.com/nsg-ethz/FANcY.

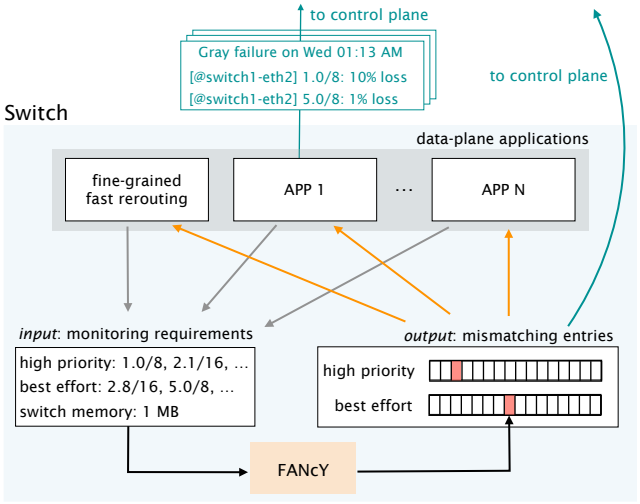


Figure 1: High-level view of a FANcY switch.

2 GRAY FAILURES IN ISP NETWORKS

We now detail why detecting and localizing gray failures in ISPs is a practically relevant, open research problem.

2.1 What do ISP network operators say?

We conducted an anonymous survey (in accordance with the directives of our university's institutional review board) amongst operators on the NANOG mailing list. The vast majority (80%) of the 46 respondents operate a WAN.

ISPs suffer from gray failures operators care about. Gray failures are an actual problem for ≈90% of the operators. Among those, 13% need to diagnose gray failures every day, 46% at least once a month, and 73% at least once every half a year. Several operators acknowledge that they typically investigate gray failures in response to customers' complaints. Though alarming, these figures likely underestimate the actual number of gray failures occurring in ISPs. Indeed, 74% of the operators indicate that they do not use any gray failure detector, implicitly confirming that they do not have practical tools for it. Hence, many (if not most) gray failures likely go unnoticed even when they do affect some customer traffic.

ISP operators struggle to diagnose gray failures. Operators are overwhelmingly clear: when discovered, gray failures are difficult, time consuming, and frustrating to debug. Such a process takes hours for 35% of the operators, days for 20%, and weeks for another 20% of the operators. This is partially motivated by the lack of tools to localize gray failures. Indeed, the most common approach to troubleshoot them consists of manually dismissing assumptions one by one. Since gray failures tend to be hardware-specific, operators often need to involve vendors in this process, which slows things down even more. Worse yet, many gray failures are never diagnosed, e.g., because they appear intermittently.

2.2 What is the impact of gray failures in ISPs?

To understand how gray failures happen in practice, we analyze hardware bug reports published by Cisco and Juniper, the two largest routing vendors in the ISP market. We find more than 150 bugs resulting in gray failures.

We classify gray failures according to *(i)* the affected forwarding entries (i.e., all or some IP prefixes); and *(ii)* the dropped traffic (all or some packets per affected entry). Our classification focuses on the effects of the gray failures (i.e., *what* is dropped, which is visible to operators), rather than their causes (i.e., *why* packets are dropped, which usually known by vendors only).

Table 1 lists representative examples of gray failures for each class. It shows that gray failures come in all shapes and forms, some leading to complete blackholes while others induce drops of very specific packets only (i.e., affecting one or a few entries).

Our survey (§2.1) confirms that our classification is representative: operators state that they have observed at least one gray failure of each type.

2.3 Why is prior work not applicable?

We now discuss why prior gray failure detection approaches do not work in ISPs, and motivate the need for a new in-switch design.

Controller-centric approaches do not scale to ISPs. The most straightforward controller-centric approach is to mirror traffic to the controller, and let it compute packet differences between consecutive hops. However, such a naive approach is not practical in ISPs, since it inflates the network load proportionally to the number of links per path, and requires the centralized system to potentially process an aggregate rate of hundreds of Tbps of traffic.

Recent work has explored more practical packet mirroring options, such as truncating copied packets [24] or mirroring specific subsets of packets only [41, 45]. Despite these efforts, packet mirroring tends not to scale well with respect to traffic: either (i) a few flows are monitored accurately and quickly; or (ii) many flows are monitored but slowly; or (iii) most flows are monitored inaccurately.

An alternative to packet mirroring is instructing switches to store measurements in sketches, which are compressed data structures extracted, decompressed, and processed by a centralized controller. For detecting and localizing gray failures, however, switches cannot arbitrarily compress measurements, but must enable controllers to reconstruct traffic at *per-entry or per-packet granularity* – which most of the sketches proposed in previous work (e.g. [27, 29, 30, 33, 34, 42]) do not do. This need prevents the data structures maintained in the switches from being arbitrarily small. In turn, reading values from relatively large sketches takes too much time for failure detection to be fast and practical for ISPs.

As an illustration, we consider Loss Radar [31], one of the few sketch-based systems able to detect gray failures. Loss Radar uses in-switch Invertible Bloom Filters (IBFs) to track XORs of packets: by comparing differences between the IBFs of consecutive switches, the controller tries to reconstruct the headers of packets lost at each hop. To ensure quick failure detection and limit IBFs' sizes, IBFs must be extracted very often (i.e., every 10 ms).

Table 2 shows the results of measurements we performed on a state-of-the-art switch: current switches do not read memory fast enough for Loss Radar to support average loss rates higher than 0.15% in 100 Gbps switches with 32 ports. Loss Radar limitations worsen for higher bandwidth.

LossRadar requirements

Switch	Metric	Average loss rate 0.1%	0.2%	0.3%	1%
100 Gbps 32 ports	memory size*	× 0.21	× 0.42	× 0.63	× 2.1
	read speedup†	× 0.7	× 1.4	× 1.9	× 4.5
400 Gbps 64 ports	memory size*	× 1.7	× 3.4	× 5.1	× 16.9
	read speedup†	× 3.7	× 6.6	× 9.5	× 29.5

* LossRadar req. memory / memory available per hardware stage

† LossRadar req. read speed / available hardware read speed

Table 2: Even for registers' (64 bits) and packets' (1500 B) sizes minimizing memory reading time, LossRadar exceeds the capabilities of state-of-the-art switches (see red numbers).

Note that switches' memory size and reading speed constraints limit the options available to sketch-based approaches: extracting measurements less frequently requires larger data structures, which however exacerbate hardware limitations. For example, in Loss Radar, gathering IBFs less frequently is counter-productive because it requires increasing their sizes to deal with the higher number of packets lost during larger intervals for the same loss rate; yet, larger IBFs further reduce the loss rates detectable by Loss Radar.

Our results show that Loss Radar fundamentally cannot detect gray failures efficiently within current and future ISPs, unless a major technological breakthrough enables switches to support significantly more memory and read it much faster than today. Other sketches may make a more parsimonious use of switches' memory, but they are likely to suffer similar scalability limitations with respect to the tracked traffic. In addition, a recent study [35] shows that all sketch-based solutions have a significant accuracy drop (up to 94×) compared to theoretical expectations due to the delays in retrieving the data plane state. Those limitations lead us to design an in-switch failure detection system.

Existing in-switch designs are not suitable for our goals. Two techniques have recently been proposed to detect failures within switches: Blink [26] and NetSeer [44].

Blink [26] focuses on failures affecting all the flows crossing a failed link. It selects a small number of flows (e.g., 64) per prefix, and checks if the majority of them retransmits within a 800 ms window. Blink fundamentally cannot detect a gray failure that does not affect the majority of the flows crossing a link: in those cases, Blink simply does not monitor enough affected flows. For cases in which Blink could select more than 32 affected flows, gray failures increase the likelihood that retransmissions are spread over time, beyond 800 ms windows, since only a subset of the packets is lost, which would again prevent Blink from detecting the failure.

Extending Blink to detect gray failures is also challenging as (i) monitoring more flows is impractical, given switches' computational and memory resources; and (ii) inferring failures from the retransmissions of fewer flows would lead to many false positives.

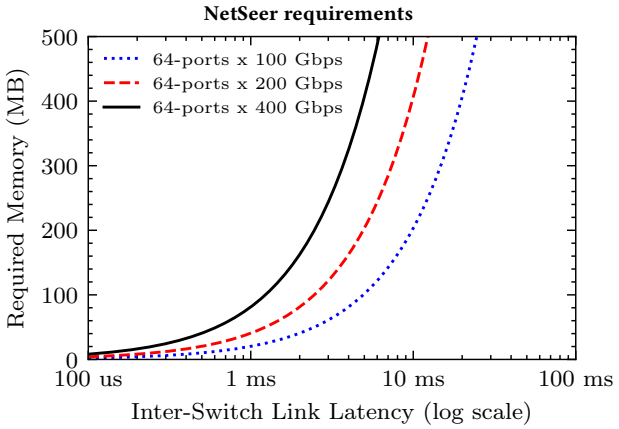


Figure 2: Total memory per switch required by NetSeer to detect and localize gray failures.

NetSeer [44] is an in-switch system designed to detect a variety of events, including gray failures, in data center networks. It includes mechanisms to identify packet drops internal to switches (e.g., caused by congestion), as well as an inter-switch protocol for detecting the most general class of gray failures.

While NetSeer keeps working for internal losses correctly logged by switches, it struggles to detect many gray failures in ISPs, because of the design of its inter-switch protocol. Indeed, in the NetSeer's inter-switch protocol, each switch stores a signature of sent packets in a buffer, and receives NACKs from neighbors whenever any such packet is lost. Fundamentally, NetSeer's packet buffers have limited size, and in ISPs, they are likely to be overridden before NACKs are received, because of ISPs' traffic volume and link delays. Whenever this happens, we say that NetSeer is not operational, since it has no visibility on losses per entry and therefore cannot localize the corresponding gray failures.

Figure 2 shows that NetSeer is not operational in the most common ISP settings, where traffic per link exceeds 100 Gbps and link latency is on the order of milliseconds – the operator survey we conducted confirms those numbers. Results in the figure are computed analytically, and confirmed by experiments we perform in ns-3. In contrast to the hundreds of MBs required by NetSeer to be operational on ISP links, memory available to in-switch applications tends to be in the order of few MBs. Indeed, current switches offer about 12-15 MB of memory per pipeline, with 4-8 pipelines in total [5]. For each pipeline, however, the available memory is split across the pipeline's stages [19, 25], meaning that an in-switch application is in practice constrained by the maximum per-stage memory. In addition, per-pipeline per-stage memory is shared across all in-switch applications, further reducing the memory available to each application.

Note that NetSeer's limitations also do not seem to be easy to fix in the future. In fact, traffic forwarded by ISPs is expected to increase over the years at a much faster pace than hardware resources (e.g., memory) in switches. This trend would make NetSeer less and less suitable for future ISPs.

2.4 What about simple designs?

An outcome of §2.3 is that a gray failure detector for ISPs should work in-switch, and neither sample flows nor keep per-packet information. At a glance, it may seem that we can use simple designs matching those constraints by just exploiting the ability of switches to count packets. Unfortunately, this is not the case.

First, we cannot count traffic at per-link granularity: this simply does not provide enough information to localize the failure. Also, we cannot sample traffic to count; otherwise, gray failures affecting small fractions of traffic would probabilistically take a long, possibly indefinite time to be even detected. Similarly, we cannot count traffic only for some entries because gray failures can impact one or a few entries that we do not know in advance – see Table 1.

Conceptually, we instead need to count *all* packets for each entry. Once again, however, simply having one counter per entry does not work in ISPs, as it exceeds the memory available on the switches' hardware. For example, if we consider entries to be IPv4 prefixes, covering the Internet routing table with 32 bits per counter would require about 512 MB on a 64-port switch.

In §3, we describe the design that we propose to scale per-entry packet counters. Such a design also addresses practical challenges, including how to distinguish gray failures from transient drops (e.g., congestion-induced), and how to ensure that switches count the same packets with the same counters.

3 FANCY OVERVIEW

Given the input entries to monitor and memory budget as shown in Figure 1, FANcY switches allocate one dedicated entry for each high-priority entry, while best-effort entries are collectively monitored through one hash-based tree.

FANcY works at a per-link granularity, reporting losses separately for each switch port. To detect and localize gray failures affecting input entries, each upstream FANcY switch sending packets to a downstream FANcY switch establishes counting sessions with the downstream, opening a new session as soon as the previous one is closed. During each counting session, the upstream tags packets to be counted by the downstream with an identifier of the counter to be increased, so that both switches consistently count the same subset of packets with the same counters. At the end of each session, the downstream sends back its counters to the upstream, which compares the counters and immediately after, starts a new session. When it detects discrepancies between its counters and the downstream ones, the upstream switch flags the mismatching counters by populating local registers.

FANcY counters are carefully positioned to avoid recording packet loss due to congestion. Within any switch, congestion typically occurs at the traffic manager (TM), which implements the actual switching logic – i.e., redirecting packets from the ingress pipeline to the configured egress pipeline. In FANcY, packets are therefore counted after the TM of the upstream switch and before the TM of the downstream one.

We design FANcY's counting protocol to be resilient to packet loss while also using minimal memory on switches. To provide good accuracy for best-effort entries, we rely on a zooming algorithm that allows switches' data planes to dynamically explore hash-based trees at runtime. This reduces FANcY's memory consumption on

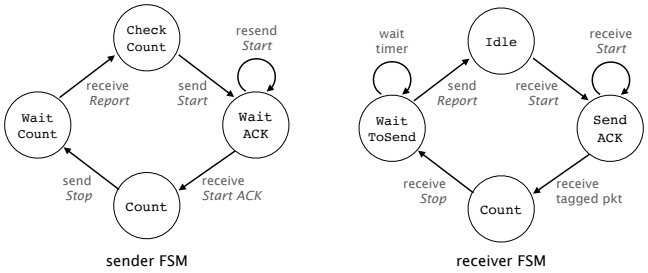


Figure 3: Finite state machines run on any pair of upstream (left) and downstream (right) FANcY switches.

switches, thus allowing each switch to maintain counting sessions with all its downstream switches. We detail the design of FANcY internals in §4, and we delve into its accuracy, speed, and resource consumption in §5 and §6.

4 FANCY INTERNALS

We now describe the most important FANcY components: the counting protocol (§4.1), the hash-based tree data structure (§4.2), and the system interface and deployment (§4.3).

4.1 Counting protocol

When designing FANcY's counting protocol, we need to balance accuracy (i.e., how often we count packets), reliability (i.e., how to guarantee that counters are successfully exchanged), and scalability (i.e., how much memory is needed on switches). We first show that maximizing accuracy leads to high memory consumption and sub-optimal reliability. This motivates us to trade some accuracy for much better reliability and scalability.

Strawman: continuous counting with in-packet session IDs. Ideally, we would like to continuously count all the packets at the upstream and downstream switches. To do so, the upstream can tag packets with a session ID, and start a new session by just changing the packets' tag. Upon receiving a packet with a different tag, the downstream would then send its counters back to the upstream.

Unfortunately, this simple approach requires the upstream switch to allocate memory for two sets of counters, respectively for the current and previous sessions. The upstream indeed has to wait for the counters from the downstream switch before it can check for packet drops in the previous session. In addition, the above protocol does not achieve reliability. If a counter sent by the downstream is lost, all the measurements for that session are also lost – i.e., a link cannot be monitored if a failure affects the reverse direction of the traffic. To ensure reliability across k sessions, both the upstream and the downstream must then keep $k-1$ historical counters' values, and consume k times the memory required for a single session.

FANcY protocol. To achieve reliability with minimal memory, FANcY adopts a protocol akin to stop-and-wait. In FANcY, every counting session is opened by the upstream switch through a *Start* control message, and closed after a *Stop* message. After sending a *Start* (resp. *Stop*) message, the upstream switch waits for a *Start ACK* (resp. *Report*, including the downstream counters) response

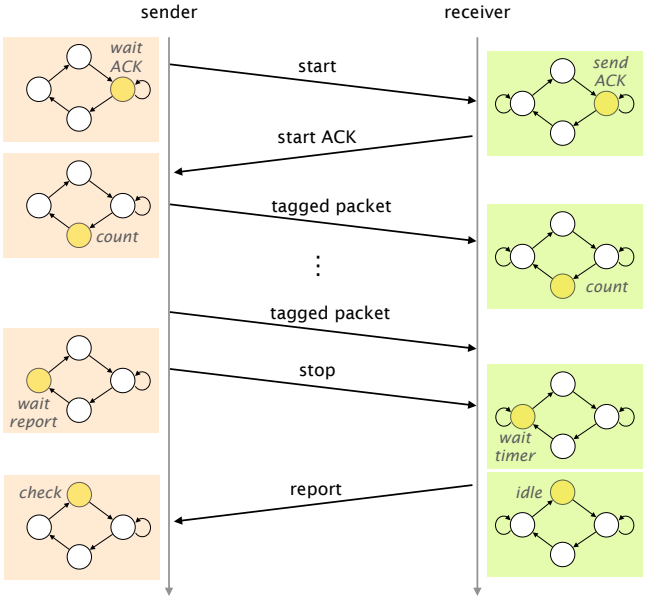


Figure 4: Time sequence diagram showing the implementation of a counting session with FANcY state machines.

from the downstream switch, and it keeps retransmitting *Start* (resp. *Stop*) messages if it does not receive responses before a timeout.

At any time, FANcY's counting protocol requires storing a single set of counters, for the current session, at both the upstream and downstream switches. Its downside is that counting is stopped when control messages are exchanged. We make this choice because FANcY focuses on systematic packet drops (e.g., see Table 1), and hence stopping counting for short times may affect detection speed, but does not prevent us from detecting gray failures, as §5 confirms.

An important parameter of FANcY's counting protocol is the frequency of counters' exchanges. This parameter influences the accuracy, the detection speed and the overhead in terms of additional traffic generated by FANcY. We discuss reasonable counters' exchange frequency values in §5.

FANcY Finite State Machines (FSMs). FANcY switches implement its counting protocol by running FSMs directly in the switches' hardware. We now detail FANcY's FSMs. We further describe the implementation and evaluation of those FSMs within an Intel Tofino switch in §6 and Appendix B.

Let A be an upstream switch, and B be the downstream one. To detect losses of packets sent by A to B, A implements FANcY's *sender FSM*, while B runs the *receiver FSM*. Figure 3 displays both FSMs, and Figure 4 illustrates how the FSMs transition from one state to another during a typical counting session.

To start a new counting session, A resets all its counters for the $A \rightarrow B$ link, and sends a *Start* message to B. Since it is critical that A and B start counting from the same packet, A then enters the `WaitACK` state where it doesn't increase any counter but waits for an acknowledgement from B. When it receives a *Start* message, B indeed resets its counters, and replies with a *Start ACK* message. If after a given time T_{rtx}, A does not receive a *Start ACK*, it sends the

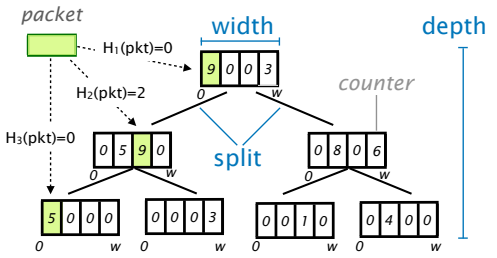


Figure 5: Example hash-based tree implemented within FANcY switches: each node is an array of counters, and packets are mapped to counters at each level through a level-specific hash function.

Start message to B again. If A does not receive responses from B after X attempts (with X=5 by default), A reports a link failure.

After receiving a *Start ACK*, the sender FSM transitions to the `Counting` state, where A counts and tags each packet it sends over the $A \rightarrow B$ link. The first tagged packet received after the *Start* message makes B transition to its own `Counting` state.

Packets are tagged by A and counted by B until A sends a *Stop* message to B. At that point, A moves to the `WaitCounter` state until it receives a *Report* message from B, with B's counters.

A stop-and-wait approach, similar to the one implemented for the session setup, is used to work around possible losses of *Stop* and *Report* messages. In contrast to session opening, however, the downstream switch does not send the *Response* message immediately after receiving the *Stop* one. Upon receiving such a message, the receiver FSM indeed transitions to the `WaitToSendCounter` state, where it can keep counting tagged packets for a short time interval T_{wait}. This timeout accounts for delayed or reordered packets. In theory, it should not be possible for packets to be reordered if they follow the exact same path from the sender FSM to the receiver one – e.g., if A and B are neighbors. We keep the `WaitToSendCounter` state in the receiver FSM to avoid making assumptions on the path from A to B.

We note that our FSMs can be easily extended to synchronize and exchange arbitrary state across switches. Indeed, exchanging information other than packet counters only requires to tweak the semantics that switches associate to packet tags, and adjust the content of the *Report* messages.

4.2 Hash-based trees

FANcY hash-based trees are a generalization of Bloom filters. Each tree level stores counters associated with a subset of best-effort entries. Counters at higher levels of the tree map to larger sets of entries, while the tree's leaves map to one or few entries. A Bloom filter can then be seen as a hash-based tree with only one level.

We use this generalization in order to achieve both accuracy and scalability, at the cost of reducing the detection speed by a factor proportional to the number of levels in the tree. For a large number of entries, Bloom filters either have a large memory footprint, or are subject to collisions and hence false positives. We sidestep from this memory-accuracy tradeoff by instructing the switches to explore hash-based trees at runtime, zooming in counters at lower levels when a failure is detected for counters at higher levels.

Hash-based trees' data structure. Figure 5 pictures a small hash-based tree as implemented in FANcY switches. Its nodes are fixed-size arrays of counters. Each counter is mapped to a specific set of packets through hash functions. To better define which packets increment which counters, we first introduce some terminology.

Any FANcY hash-based tree is a balanced k-ary tree, characterized by three parameters: width, depth, and split. For a given tree, its *width* w is the number of counters per node, its *depth* d is the length of any path from the tree's root to a leaf, and its *split* k is the number of children per node. For example, $w = 4$, $d = 3$, and $k = 2$ for the tree in Figure 5.

Every packet belonging to a best-effort entry maps to one counter per tree's level through a distinct hash function per level, as also shown in Figure 5. Consider a counter c_i. A packet p is mapped to c_i if and only if $H_j(p) = i$, where H_j is the hash function applied at the level j to which c_i belongs, and $H_j(p)$ is a value between 0 and $w - 1$ obtained by applying H_j to the fields of p used in p's entry (e.g., the destination address in destination-based routing).

We define the *hash path* of a packet as the list of counter IDs the packet maps to, ordered from the root to the leaf. For instance, $[0, 2, 0]$ is the hash path of the packet shown in Figure 5.

Note that any sequence of counters at consecutive levels forms a *partial hash path*, and corresponds to a number of entries inversely proportional to the length of the sequence: the shorter the sequence, the bigger the number of associated entries.

FANcY zooming algorithm. To detect best-effort entries affected by a failure, this algorithm incrementally builds partial hash paths of increasing length for counters affected by a failure. At every iteration, the algorithm indeed increases by one the length of the partial hash path affected by packet loss, and hence it reduces the set of candidate failed entries.

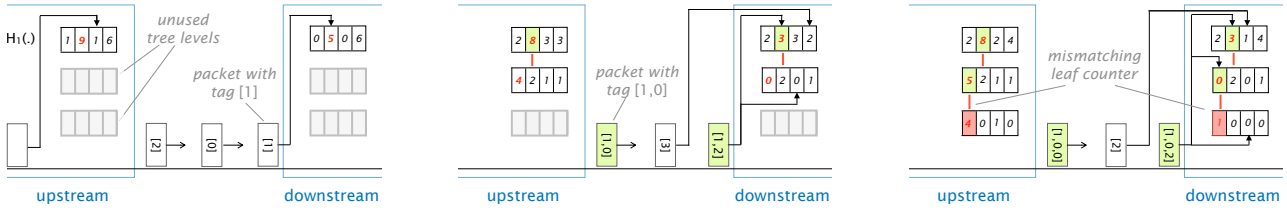
Consider a pair of FANcY switches. Assume for now that the switches maintain trees of split 1, as shown in Figure 6.

In the absence of losses, the two switches only update root-level counters. During each counting session, the upstream switch tags every packet with the index of the counter to which the packet maps according to the root-level hash function, and the root-level counters are consistently updated on both switches, as displayed in Figure 6a. At the end of the session, if no drops have happened, the upstream switch detects the congruence of its counters with the downstream ones, and starts a new session.

Suppose now that a gray failure occurs. At the end of the session, the upstream switch checks its counters against the downstream ones. If it detects mismatches for more than half of the counters, it flags the failure as a uniform random one – i.e., "localizing" it to all entries. Otherwise, the switch computes the root-level counter c_i with the maximal difference between the local and the downstream values.[1] In the following session, the upstream switch then tags packets if and only if they hash to c_i, effectively zooming in the set of entries with the highest drop rate.

Packet tags carry information about the hash path of the counters packets map to. This way, the downstream switch knows which

[1] Selecting the counter with maximum losses is instrumental to prioritize failure detection for most traffic. We however envision that future FANcY implementations can take operators' policies into account at this step.



(a) before any counter mismatch (b) counting session after first zoom (c) successful zooming completion

Figure 6: Illustration of the zooming algorithm on hash-based trees of width 4, depth 3 and split 1.

packets to count and which counters to increase without having to hash packets consistently with the upstream. In our example, after detecting a mismatch on counter c_i, the upstream would therefore tag every packet that maps to c_i with the path $[i, m]$, where i is the index of c_i in the root-level node and m is the index of the second-level counter c_m to which the packet is mapped – see Figure 6b.

The above procedure is repeated until a leaf node is reached. At that point, FANcY reports a failure for *every* mismatching leaf counter (see Figure 6c). This, for example, enables the upstream switch to immediately start rerouting packets whose hash path corresponds to any of those counters.

Note that FANcY technically detects a failure when it starts zooming in any root-level counter, but reports the failure only after reaching the tree's leaves in order to increase accuracy.

For multi-entry failures, FANcY adopts a pipelining approach that increases failure detection speed. To achieve that, it simultaneously zooms in counters at different levels of the tree. Consider, for example, a failure that affects two root-level counters c_1 and c_2. At the end of the first counting session after the failure, the upstream switch observes packet losses on both c_1 and c_2, and selects the one with the maximum packet difference, say c_1. In the following session, the upstream then instructs the downstream to populate counters at the second level of the tree for packets hashing to c_1 *in addition* to increasing root-level counters. At the end of this second session, the upstream observes again packet loss for both c_1 and c_2. Since it is already zooming in c_1, it starts zooming in c_2 this time. So, in the third session, the upstream and downstream increase root-level counters for all the packets, second-level counters for packets hashing to c_2, and third-level counters for packets hashing to c_1 and the second-level counter with the maximum mismatch in the previous session.

A generalized version of the above algorithm is used in trees with split $k > 1$. At the end of every counting session, the generalized algorithm zooms in k mismatching counters rather than only one. In the presence of multi-entry failures, the algorithm therefore can explore *in parallel* up to k hash paths per counting session, and hence supports the simultaneous exploration of k^{d-1} different paths in d counting sessions.

Additional properties and analyses of the tree's parameters when used in combination with the above zooming algorithm are reported in Appendix A.

4.3 Practical considerations

We now describe how FANcY design is instantiated.

Input translation. FANcY switches first allocate one dedicated counter for each input high-priority entry. Each of those counters occupies 80 bits in total (both at upstream and downstream), including the required state for the counting protocol.

Switches then dimension the hash-based tree based on the input memory minus the amount consumed by dedicated counters. Each node of the tree requires at each side of the session 32 bits times the width of the tree, plus 88 bits to support the counting protocol and the zooming algorithm. Appendix B provides more details.

The question then is how to decide the width, depth, and split of the tree, which also influences the number of nodes. We analyze the impact of those parameters on the system performance performing a sensitivity analysis based on CAIDA traffic traces (see Appendix D). Our analysis indicates that setting split to 2 and depth to 3 provides a good tradeoff between memory consumption, accuracy, and detection speed. Hence, our FANcY implementation uses those values, and adjusts the tree's width on the basis of the available memory. ISP operators can customize FANcY's trees by applying a similar analysis on their networks' traffic traces.

FANcY returns an error if the memory needed for dedicated counters and hash-based tree with the above parameters' values exceeds the input memory.

Output. FANcY uses two additional data structures to flag the entries affected by packet loss: a 1-bit register array with one register for each dedicated counter, and a 2-register Bloom filter associated with the hash-based tree. When mismatching values are detected for a dedicated counter, the corresponding register in the 1-bit array is updated. When a counter in the hash-based tree reports a failure, the hash path for that counter is stored in the Bloom filter.

Deployment. FANcY is designed to be deployed at every switch, so that it can monitor all links, one by one; this maximizes accuracy of failure detection and localization.

We however note that FANcY keeps working when deployed at remote switches. In this case, FANcY is able to detect gray failures on the *path* between the two switches[2], although losing the ability to precisely pinpoint the failure location along the path. This enables practical use cases in partial and incremental FANcY deployments. For example, if deployed at the border switches exchanging high volumes of traffic, FANcY provides support for near real-time

[2]Note that systematic failures can be distinguished from congestion even in partial deployments of FANcY by monitoring queue sizes on all devices, and discarding all measurements collected during periods where queue sizes were excessively long.

detection of gray failures along the internal paths carrying most traffic: no tool currently available to ISPs offers a similar capability.

5 EVALUATION

We evaluate FANcY against its goal of detecting and localizing gray failures accurately, quickly and scalably. Since FANcY is a data-driven solution, its accuracy and detection speed depend on the amount of traffic it receives for the entries affected by gray failures. We therefore assess FANcY's performance depending on the packet loss rate per disrupted entry. We do not compare against gray failure detectors proposed in previous work because they are incompatible with the network characteristics of ISPs, as already detailed in §2.3.

To evaluate FANcY, we simulate different gray failures and measure FANcY's accuracy and speed to localize each of them. We use synthetic traffic to quantify the minimal requirements for FANcY to properly work (§5.1), and CAIDA traces [21] to evaluate the system-wide performance on real traffic (§5.2). Finally, we analytically assess FANcY's scalability in terms of traffic overhead (§5.3).

In our evaluation, we consider a 64-port FANcY switch which is given the following input: high-priority entries covering the 500 prefixes driving the most traffic, best-effort entries for all the remaining traffic, and memory of 1.25 MB (i.e., 20KB per port). Accordingly, FANcY uses 500 dedicated counters and a hash-based tree of depth 3, split 2, and width 190.

When evaluating FANcY's accuracy, we mainly refer to its true positive rate (TPR), which is defined as the fraction of the correctly identified failed entries. Hence, the TPR measures FANcY's ability to detect and localize failures. We focus on the TPR because the true negatives are the complement of the true positives in our case, and the false positives (i.e., entries detected as failed despite they are not) do not depend on traffic conditions. Indeed, the false positive rate (FPR) is always zero for any dedicated counter. Also, for the hash-based tree, the FPR depends on the probability that multiple entries are stored in the same leaf node, and one of them experiences losses, otherwise is zero too. This probability is a function of the tree's width and depth (as detailed in Appendix A), and is very low for reasonably dimensioned trees. In fact, for traffic extracted from CAIDA traces, the average number of FANcY's false positives is 1.1 (resp., 0.59) for 100% (resp., 1%) packet loss in the challenging case of 100 entries failing at the same time.

We measure FANcY's detection speed as the difference between the time a gray failure is introduced in an experiment and the time FANcY localizes it. Note that this is slightly unfair to FANcY as it may have to wait some time before a packet affected by the failure is received, especially if the corresponding entry drives little traffic, or the gray failure has a low packet drop rate per entry.

To show that FANcY works in large ISPs, we set the inter-switch delay to 10 ms in all the experiments. We also experiment with lower link delays, for which FANcY's accuracy slightly increases for low-drop scenarios, and failure localization speeds up. For example, for 1 ms links, detection speed doubles for dedicated counters, and increases by ≈ 15% for hash-based trees.

Experiments in this section are packet-level simulations performed with ns-3 [16]. Simulated networks are composed by nodes running our software implementation of FANcY– i.e., ≈ 8,000 lines of C++ code implementing a custom ns-3 switch that closely mimics all the data-plane components (parsers, ingress, egress, metadata fields, etc.) of a P4 switch.

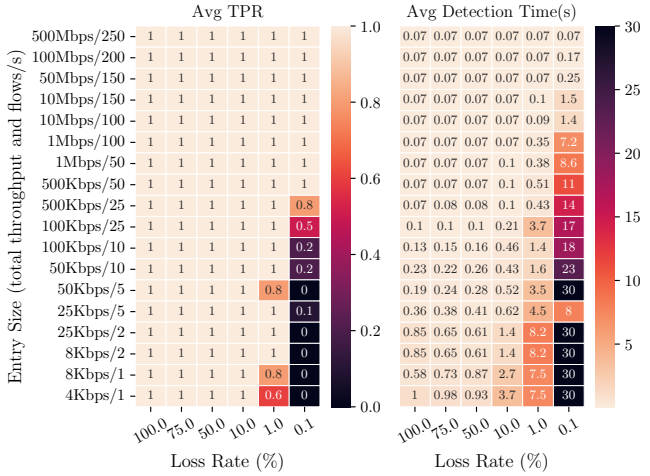


Figure 7: Accuracy and detection speed of dedicated counters for different gray failures and traffic volumes.

5.1 Benchmarking FANcY

First, we experimentally show that FANcY requires an amount of traffic per entry which is realistic to assume in ISP networks.

We are especially interested in the *minimum* amount of traffic needed for FANcY to detect different types of failures. We, therefore, evaluate FANcY's accuracy and speed for synthetically generated traffic of increasing size: in separate experiments, we generate traffic with a different number of TCP flows per second and bitrate per flow. All simulated flows have a duration of ≈1 second in the absence of losses, and a retransmission timeout of 200 ms. Of course, failures can significantly increase the duration of flows.

Within the first two seconds of each experiment, we simulate a failure by instructing a switch to drop a certain percentage of packets for some or all entries. We then run each experiment for 30 seconds. When we do not detect any failure across all the repetitions of an experiment, we report a TPR of 0 and a detection time of 30 seconds. We repeat every experiment 10 times, randomly changing flows' starting and failure times.

In the following, we first consider gray failures affecting a subset of entries monitored by FANcY, such as in the cases shown in the first row of Table 1. We do so separately for the dedicated counters (§5.1.1) and the hash-based tree (§5.1.2). We then evaluate FANcY's performance upon failures affecting all entries (§5.1.3), such as link-level problems or bugs exemplified in the second row of Table 1.

5.1.1 Dedicated counters. We assess performance of dedicated counters by simulating single-entry failures only, because those counters work independently from each other.

We first evaluate the impact of the exchange frequency of counters. In principle, such a frequency may affect FANcY's accuracy because packet losses are not detected when counting sessions are opened and closed. Our simulations, however, indicate that FANcY's accuracy is not significantly impacted unless counters

are exchanged extremely often. Accuracy results are indeed very similar whenever counters' exchange frequency ranges between 50 and 100 ms. This also means that the counters' exchange frequency just affects overhead and detection speed: increasing the exchange frequency speeds up failure detection but increases the overhead. Hereafter, we report results for a frequency value of 50 ms.

We now focus on the FANcY's performance for different traffic volumes and loss rates, mostly referring to Figure 7.

Accuracy. As displayed in the left part of Figure 7, FANcY's dedicated counters detect almost all gray failures whenever the induced packet drop rate is ≥1%, or the affected entries drive at least 500 Kbps of traffic.

Accuracy decreases for very low drop rates (e.g., 0.1%) of entries attracting little traffic (≈100 Kbps or less). However, this is not an intrinsic limitation of FANcY, but mostly an artifact of our experiments. Indeed, very few packets are generated during these experiments, and chances are low that any packet is dropped if the loss rate is ≤0.1%. For example, in 80% of those experiments, no packet is actually dropped during the 30 seconds of the experiment. Only in the remaining 20% of the cases at least one packet is dropped. In those latter cases, FANcY fails to detect simulated failures because packets are dropped while FANcY closes a counting session or opens a new one. In real deployments, operators can reduce those cases by decreasing the counters' exchange frequency, which would trade detection speed for higher accuracy in very low drop-rate scenarios.

Detection speed. For dedicated counters, we expect a failure to be detected just after the first post-failure counters' exchange. The right part of Figure 7 shows that this is the case as long as the failed entries drive enough traffic (e.g., at least 500 Kbps). In fact, the top-left part of the right heatmap shows that the average detection time is ≈70 ms, which is approximately the counters' exchange frequency (50 ms) plus counting sessions' opening and closing.

Results may look less intuitive in the bottom part of the heatmap, where the average detection time increases to ≈600-1000 ms for blackholes, and to several seconds for lower packet-drop rates. Again, this does not directly depend on FANcY. Instead, packets affected by a failure tend to appear some time after the failure is introduced for low-traffic entries and low drop rates. For example, if an entry drives one packet per second, on average the first packet for that entry is received FANcY 500 ms after the failure is introduced.

5.1.2 Hash-based tree. Contrary to dedicated counters, the performance of hash-based trees generally depends on the number of entries failing simultaneously. In fact, the detection of one failed entry may be delayed or even overlooked when FANcY zooms in the counters for another entry. We therefore evaluate both single-entry and multi-entry failure scenarios.

As the first step, we need to decide the duration of the counting sessions, which we denote as *zooming speed* for brevity. To do so, we measure the minimum prefix size required to get a TPR of at least 95% when we vary the loss rate and zooming speed. Results are plotted in Figure 8. All zooming speeds between 10 and 200 ms reach high TPR values, even for low loss rates (up to 0.1%), as long as the prefixes drive a reasonable amount of traffic. Additionally, requirements on the traffic per entry are very similar across zooming speeds higher than 50 ms.

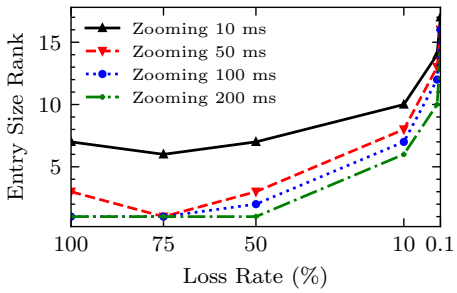


Figure 8: Minimum entry size for which FANcY has a TPR ≥95% for different zooming speeds. The y axis ranks entries according to the traffic they drive: lower ranks correspond to smaller traffic.

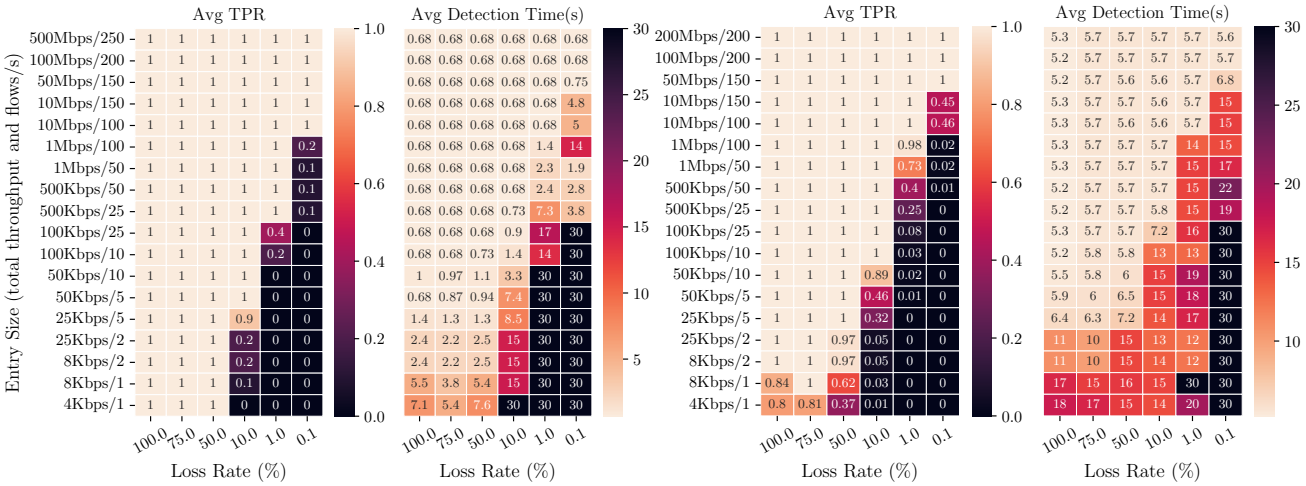
We conclude that FANcY's accuracy is not very sensitive to the tree's zooming speed between 50 ms and 200 ms. In the following, we show the results obtained using 200 ms as zooming speed, as it matches the typical value of TCP flows' retransmission timeout. We note that operators can fine-tune FANcY's zooming speed according to their specific requirements, as faster zooming speeds tend to decrease detection time but also increase overhead.

We now focus on FANcY's performance, comparing failures affecting only one entry with those impacting 100 entries at the same time. We refer to Figure 9.

Accuracy. For single-entry failures, FANcY always identifies the failed entry as long as the packet loss rate is higher than 10%. For lower loss rates, FANcY's accuracy worsens for low-traffic entries. This is a direct consequence of our design: FANcY fully detects a failure after observing packet loss in three consecutive counting sessions, which becomes unlikely if it receives few failure-affected packets. Indeed, in 97.5% of the experiments where FANcY fails to detect simulated failures, at no time are packets dropped during three consecutive counting sessions. We expect entries with those characteristics to collectively account for a limited percentage of real ISPs' traffic (see also §5.2), which makes this limitation not critical in real deployments.

For multi-entry failures, TPR values are consistent with those for single-entry failures, as evident when comparing the left part of Figure 9b with the left part of Figure 9a. TPR decreases only for very low-traffic entries (e.g., 4-8 Kbps). For 80% of the runs in which FANcY fails to detect failures, no packets are dropped during three consecutive counting sessions – with packets lost while FANcY zooms in another entry in the remaining 20% of the experiments. Again, we expect entries attracting so little traffic to be not critical for ISPs. Those results thus suggest that FANcY's trees should be able to cover the practically relevant entries in ISPs' switches, even for failures simultaneously affecting a hundred entries.

Detection speed. FANcY is fast to detect failures of single entries with a reasonable amount of traffic and high loss rates: as shown by the right part of Figure 9a, single-entry failures are typically detected in 680 ms, which roughly matches the lower bound of three times the selected zooming speed (i.e., 200 ms). FANcY detection slows down lower-traffic entries and low loss rates: for example, it changes from sub-second to a few seconds for single entries attracting ≤50 Kbps of traffic.



(a) Single-entry failures **(b) 100-entry failures**

Figure 9: Accuracy and detection speed of FANcY's hash-based tree for different gray failures and traffic volumes.

Increasing the number of failed entries has a more significant impact than the entry size. For 100-entry failures, the average detection time increases from 600 ms to about 5.3-5.7 seconds for high-loss high-traffic entries. This increase is motivated by the fact that FANcY zooms in a limited number of counters in each counting session – e.g., one root-level counter per session. This choice enables FANcY to scale, but intrinsically degrades detection speed for many-entry failures, which we believe are relatively uncommon. We also stress that for all the scenarios where FANcY has high accuracy, the detection speed remains around 5-10 seconds, which is significantly much faster than the days or weeks currently needed by most operators.

5.1.3 Uniform failures. We finally simulate failures affecting all entries simultaneously, such as random packet losses over a link, or bugs affecting all IP prefixes in Table 1. To be realistic, we simulate a network with 100 Gbps links, and assign traffic to entries mimicking a Zipf distribution. We experiment with packet loss rates per entry between 100% and 0.1%.

In all our experiments, FANcY detects the introduced failures and correctly identifies them as uniform random drops. Its average detection time matches one zooming interval (200 ms). This is consistent with the procedure used in FANcY to detect uniform failures, which is based on checking if the majority of root-level counters in the hash-based tree have mismatching values (as detailed in §4).

5.2 FANcY on real traffic traces

We now evaluate FANcY on the CAIDA traces detailed in Appendix C. The goal is to assess the traffic coverage provided by the whole system, combining the dedicated counters and hash-based tree, when traffic per entry follows a realistic distribution.

We stress that CAIDA traces constitute a challenging test for FANcY that we do not expect to be matched in real ISPs, for two reasons. First, the overall traffic rate (4-6 Gbps) in CAIDA traces is two orders of magnitude lower than typical rates in ISPs' links. Second, we assume that FANcY switches hold one forwarding entry for each /24 prefix observed in the trace (on average ≈250K), because IP addresses in the traces are anonymized at the /24 prefix granularity [14]. However, this assumption artificially inflates the number of entries with little traffic, which are exactly the ones more challenging for FANcY (see §5.1). As a reference, ≈ 60% (versus the 100% in our experiments) of the prefixes currently advertised in the Internet are /24s, according to public BGP data [15].

We rely on CAIDA traces because we are not aware of better publicly available ISP traffic traces. We however expect that in current ISPs and even more in future ones, FANcY's performance be better than the already good results it achieves in the below experiments because FANcY's accuracy and speed generally improve with higher traffic per entry – see also §5.1.

Methodology. For each CAIDA trace, we assign a dedicated counter to each of the 500 prefixes with the most bytes during the entire trace (1h), mimicking an allocation based on historical data. Then, we randomly select a 30-second slice from each trace. Note that the prefixes carrying more traffic during each slice do not generally coincide with those covered by dedicated counters.

We then implement a traffic generator that closely reproduces any input slice. In the absence of failures, the generator re-injects all the packets of each flow exactly when they appear in the slice, preserving the bit rate, packet rate, and RTT of flows. The generator relies on ns-3's TCP implementation, enabling us to run closed-loop experiments, with TCP sources reacting to packet losses.

We use each slice to perform experiments simulating the failure of the top 10,000 prefixes (which carry ≥ 95% amount of the total traffic in the entire trace), one by one, at a random time. For each prefix and loss rate, we repeat the experiment 3 times, with the time of the failure changing in each repetition. As for the simulations in §5.1, there is no guarantee that packets for the failed entries are actually dropped within the duration of the experiment, especially for low drop rates.

Loss Rate	TPR Bytes	TPR Prefixes			Detection time
		Total	Dedicated	Hash-Tree	
100%	91.3%	84.5%	100%	83.6%	2.03s
75%	96.0%	90.9%	100%	90.3%	2.59s
50%	98.7%	93.1%	100%	92.6%	2.65s
10%	96.5%	72.8%	100%	71%	4.96s
1%	77.5%	19.5%	98.9%	14.7%	8.91s
0.1%	56.6%	5%	86.7%	0.1%	6.29s

Table 3: Average accuracy and detection speed of FANcY over four CAIDA traces (see Appendix C).

FANcY's performance. As shown in Table 3, FANcY detects between 91.3% and 98.7% of affected bytes in 2-5 seconds when the loss rate is ≥10%. For the same failure scenarios, the TPR in terms of detected entries is 72.8%-93.1%, a bit lower than the TPR in terms of bytes: this happens because traffic per prefix is very skewed in CAIDA traces.

For loss rates ≤1%, FANcY's accuracy is significantly impacted (5%-19.5%), mainly because the hash-based tree's TPR decreases sharply, in line with the results presented in §5.1. The main reason for those low TPR rates is the lack of packet drops during three consecutive counting sessions, which directly prevents FANcY's failure detection in ≈80% (resp., >99.8%) of the experiments with a loss rate of 1% (resp., 0.1%). Those results further stress the importance of the hash-based tree in our design: FANcY covers only 56.6% of the bytes affected by failures when the tree's TPR is close to zero versus ≈ 99% when the tree's TPR is high.

It may seem surprising that FANcY does not perform at best when traffic is blackholed (100% loss rate). This is because FANcY measures packet loss on the observed traffic, and a hard failure immediately slows down *all* the TCP flows, reducing all affected traffic to just retransmissions. Namely, for each flow, FANcY receives the first retransmission after the expiration of the TCP retransmission timeout (typically 200 ms), and further retransmissions at exponentially increasing times. In other words, TCP congestion control makes it more likely for FANcY not to receive packets for the failed entries in three consecutive counting sessions, thus reducing the tree's TPR. In contrast, FANcY performs very well when the loss rate is around 50%, where TCP reduces the flow rate much less significantly and less abruptly.

Comparison to baselines. We compare FANcY's results with the simpler designs outlined in §2.4: a single counter per link, and one dedicated counter for each prefix.

Both designs achieve a slightly higher accuracy than FANcY: their TPR for prefixes is ≈97-99.6% for a loss rate ≥10%, ≈84% for a loss rate of 1%, and ≈35% for a loss rate of 0.1%. Their accuracy is not 100% because switches may not receive traffic for the failed entries before our experiments terminate, and may not detect packet losses when exchanging counters.

However, a single counter cannot localize any failure; the number of false positives in each experiment is the total number of prefixes minus the failed ones – i.e., ≈250K. In contrast, the solution with one dedicated counter per entry has zero false positives, but it requires 320 MB (including support for the counting protocol) versus the 1.25 MB consumed by FANcY in total. Note that the memory required by one dedicated counter per entry is expected to be ≈4 times in real ISPs holding the full BGP table (i.e., ≈900K prefixes).

We then consider two additional alternatives compatible with FANcY's memory usage. The first alternative is to allocate only dedicated counters but without exceeding FANcY's memory budget. With 1.25 MB, we can allocate a maximum of 1,024 dedicated entries per port. This approach is accurate and fast for the covered prefixes, but detects no failure for any of the remaining ≈249K ones, which carry ≈40% of the traffic in the considered CAIDA traces. As the second alternative, we consider allocating all the memory to a counting Bloom filter. The TPR of such a Bloom filter is largely consistent with the single-counter approach. However, for each detected single-entry failure, the Bloom filter reports ≈100 false positives versus the ≈0.03 of FANcY. Once again, we expect that the number of false positives for the Bloom filter solution be much higher in real ISPs, where switches typically hold significantly bigger routing tables.

Takeaways. Our results confirm FANcY's ability to detect different types of gray failures, covering the vast majority of the real-world traffic, while also achieving a much better tradeoff between accuracy, speed, and scalability than simple designs.

We expect FANcY to perform significantly better when deployed in actual ISPs. Indeed, CAIDA traces contain unrealistically low traffic per entry with respect to current and future ISP settings – a condition unfavorable to FANcY as already demonstrated in §5.1.

Results in this section are consistent with those for synthetic, non-bursty traffic, described in §5.1. They also provide consistent indications on the limits of FANcY: tiny failures of entries driving little traffic tend to be very hard to detect with hash-based trees. If operators want to protect specific entries from low loss rates, one option within FANcY's design is to specify them as high-priority entries in the FANcY's input.

5.3 Overhead analysis

We now show that FANcY's overhead is very limited on ISP-scale links. In FANcY, we have two overhead components: control packets (including counters) and packet tags added by FANcY switches.

We first consider the overhead of control packets. For dedicated counters, FANcY sends five minimum-size packets (e.g., 64 B Ethernet frames) for each link and each counting session. With 500 dedicated counters exchanged every 50 ms on a 10 ms delay link, FANcY uses ≈0.014% of a 100 Gbps link's capacity. For hash-based trees, FANcY also exchanges five control packets, including the hash-tree counter that carries 5320 B in the pipelined version of the zooming algorithm. The resulting traffic overhead is ≈0.00017% on 100 Gbps links for a zooming speed of 200 ms.

To tag packets, FANcY needs 2 bytes to specify the counter ID on each packet matched by a dedicated counter. The same amount of bytes are added to packets counted in the hash-based tree, where one byte encodes the hash path of the tree's node, and the other byte identifies the counter within the node. The tagging overhead is therefore 0.13% on a 1,500 B packet. Note that tags can also be encoded in unused header fields, which would lead to zero overhead.

Resource	Dedicated Counters	Full FANcY	FANcY + Rerouting	switch.p4
SRAM	4.80%	6.65%	8.1%	29.58%
Statefu ALU	16.66%	27.08%	33.33%	14.58%
VLIW Actions	9.4%	14.1%	15.6%	36.72%
TCAM	1.4%	2.1%	2.1%	32.29%
Hash bits	5.8%	11.8%	13.1%	34.74%
Ternary Xbar	1.8%	3.10%	3.10%	43.18%
Exact Xbar	5.1%	10.8%	12.3%	29.36%

Table 4: Hardware resource usage of FANcY compared to the baseline switch.p4 on a 32-port Intel Tofino switch.

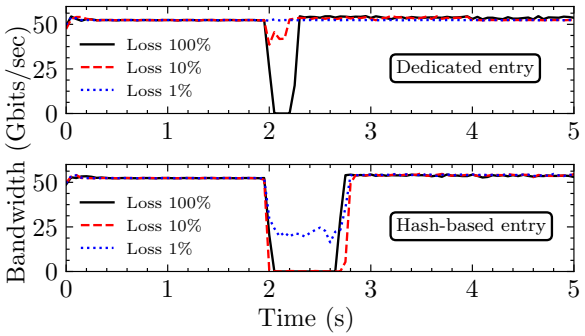


Figure 10: Case study using our FANcY implementation on a Tofino switch: FANcY detects gray failures even affecting only 1% of the packets per entry, and reroutes the traffic only for the affected entries in less than one second.

6 TOFINO MADE FANCY

We implement FANcY in ≈3200 lines of P4 code running on an Intel Tofino switch [18] with 32 ports. Our implementation is detailed in Appendix B.

Hardware resource usage. Table 4 summarizes the resource usage of FANcY, using switch.p4 as a baseline. Overall, FANcY uses a modest amount of hardware resources, including only 6.65% of SRAM (8.1% with rerouting). Stateful ALUs are the only resource that FANcY uses more than switch.p4: this is because FANcY performs several stateful operations to support counters and the counter exchange protocol. For more details about SALUs usage, see Appendix B.2. Note that SRAM is the only resource that increases when FANcY is given a higher memory budget and then uses more dedicated counters or larger trees.

6.1 Case study: fine-grained fast rerouting

As a case study, we build an application on top of FANcY that reroutes packets as soon as the corresponding counters are flagged as mismatching by FANcY (see §4.3). Note that simply rerouting might not be enough to fix some of the problems shown in Table 1. However, it might be enough for inter-switch gray failures caused by faulty links. We now detail our experiments with this application.

Setup. We use two servers, a sender and a receiver, and two Wedge 100BF-32X [18] Intel Tofino switches. The servers are equipped with Intel Xeon E5-2670 v3 2.30GHz CPUs, 256 GB of RAM, and a Mellanox ConnectX-5 100 Gbps NIC.

We connect each server to the Tofino switch that runs FANcY. The sender server generates TCP flows for a total of 50 Gbps of traffic, and 50 Mbps of UDP traffic. For each port, the FANcY switch maintains 500 dedicated counters, and implements a hash-based tree of depth 3, split 1, and width 190. Dedicated counters are exchanged every 200 ms, and the zooming speed for the tree is set to ≈200 ms. We run separate experiments for prefixes mapped to dedicated counters, and for those covered by the hash-based tree.

We use the second Tofino switch as a *link switch* connecting two ports of the FANcY switch. After 2 seconds from the start of each experiment, we instruct the link switch to drop 1%, 10%, or 100% of the packets (in different experiments). We also deploy a third link between the FANcY switch and the link switch, to provide the former with a backup next-hop.

Experimental results. Figure 10 shows the traffic throughput as measured in our experiments. In each experiment, the FANcY switch always detects the failure event less than one second after it is introduced, even when the drop rate is only 1%, and the affected traffic is monitored by the hash-based tree. As expected, the detection time is proportional to three times the zooming speed (here, 3 × 200 ms) when failures affect entries covered by the hash-based tree. On the other hand, dedicated counters ensure a predictable detection time which only depends on the counting session duration (here, 250 ms). Note that we have used a relatively higher counting session duration than the one used during the evaluation (50 ms) so that impact of the failure is noticeable in the plot.

7 CONCLUSIONS

We introduced FANcY, a data-plane system designed to detect intra-domain gray failures in ISPs. FANcY enables switches to synchronize counters in a reliable and scalable way. Its counter-based architecture complements pre-existing failure detection approaches, which are effective in data centers but do not scale to high traffic volumes and link delays.

Although FANcY focuses on detecting and reporting (but not directly fixing) failures, its interface enables future applications such as selective fast rerouting or root cause analyses. As a feasibility proof, we implemented a prototype of FANcY in a commercial Tofino switch, and demonstrated how our implementation enables sub-second fast rerouting around gray failures.

Our evaluation shows that FANcY can detect and localize gray failures quickly and accurately in ISP settings, except those that induce few, sporadic packet losses per entry – as expected, since FANcY is a data-driven system. For the very same reason, we stress that FANcY performance *improves* at higher traffic volumes, which makes its design future-proof.

Ethical issues. This work does not raise any ethical issues.

ACKNOWLEDGEMENTS

We would like to thank our shepherd Henning Schulzrinne and the anonymous reviewers for their insightful comments. We also thank Rüdiger Birkner and Romain Jacob for their comments on earlier versions of the paper.

REFERENCES

[1] Cisco Bug: CSCea91692 - PSA has a corrupted cef entry, affecting IP:IP traffic. https://quickview.cloudapps.cisco.com/quickview/bug/CSCea91692.

[2] Cisco Bug: CSCtc33158 - 7600-ES+40G3CXL drops random sized L2TPv3 packets with cookies enabled. https://quickview.cloudapps.cisco.com/quickview/bug/CSCtc33158.

[3] Cisco Bug: CSCti14290 - VPN Aggregate Label dmac corruption in hardware forwarding entry. https://quickview.cloudapps.cisco.com/quickview/bug/CSCti14290.

[4] Cisco Bug: CSCuv31196 - Random MPLS Packet Drops With IP Multicast Over L3 Ring on ASR901. https://quickview.cloudapps.cisco.com/quickview/bug/CSCuv31196.

[5] Intel tofino 3 brief. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[6] Juniper Bug: PR1296089 – Traffic received from core are not sent to locally attached circuit due to QSN timeout. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/18.1/jd0e17997.html.

[7] Juniper Bug: PR1309613 – Traffic loss may be seen if sending traffic via the 40G interface. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/17.4/jd0e19328.html.

[8] Juniper Bug: PR1313977 – Traffic drop occurs on sending traffic over "et" interfaces due to CRC errors. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/17.4/jd0e19328.html.

[9] Juniper Bug: PR1398407 – On SRX4600 and SRX5000 line of devices, BGP packets might be dropped under high CPU usage.. (Open Registration Required). https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1398407.

[10] Juniper Bug: PR1434567 – IPv6 neighbor solicitation packets getting dropped on PTX. (Open Registration Required). https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1434567.

[11] Juniper Bug: PR1441816 – Egress stream flush failure and traffic blackhole might occur (Open Registration Required). https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1441816.

[12] Juniper Bug: PR1450545 – Traffic loss might occur when there are around 80,000 routes in FIB (Open Registration Required). https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1450545.

[13] Juniper Bug: PR1459698 – Silent dropping of traffic upon interface flapping after DRD auto-recovery (Open Registration Required). https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1459698.

[14] Summary of anonymization best practice techniques. hhttps://www.caida.org/projects/predict/anonymization/.

[15] Visibility of ipv4 and ipv6 prefix lengths in 2019. https://labs.ripe.net/Members/stephen_strowes/visibility-of-prefix-lengths-in-ipv4-and-ipv6.

[16] Network Simulator 3., 2018. https://www.nsnam.org/.

[17] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 419–435, 2018.

[18] Barefoot. Barefoot Tofino, World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/.

[19] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, pages 313–323. IEEE Computer Society, 2018.

[20] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002.

[21] CAIDA. The CAIDA UCSD Anonymized 2013/2014/2015/2016/2018 Internet Traces. http://www.caida.org/data/passive/passive_2013_dataset.xml.

[22] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004. http://www.ietf.org/rfc/rfc3954.txt.

[23] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David a Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. 45:139–152, 2015.

[24] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, pages 71–85, 2014.

[25] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with p4all. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 168–174, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[27] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.

[28] D. Katz and D. Ward. Bidirectional Forwarding Detection. RFC 5880, 2010.

[29] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003.

[30] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):145–156, 2006.

[31] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 481–495. ACM, 2016.

[32] Stephane Litkowski, Ahmed Bashandy, Clarence Filsfils, Pierre Francois, Bruno Decraene, and Daniel Voyer. Topology Independent Fast Reroute using Segment Routing. Internet-Draft draft-ietf-rtgwg-segment-routing-ti-lfa-08, Internet Engineering Task Force, January 2022. Work in Progress.

[33] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[34] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[35] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas SekaR, and Peter Steenkiste. *Telemetry Retrieval Inaccuracy in Programmable Switches: Analysis and Recommendations*, page 176–182. Association for Computing Machinery, New York, NY, USA, 2021.

[36] Peter Phaal, Sonia Panchen, and Neil McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational), September 2001. http://www.ietf.org/rfc/rfc3176.txt.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive Realtime Datacenter Fault Detection and Localization. *Nsdi*, pages 25–30, 2017.

[38] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[39] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. D2R: policy-compliant fast reroute. In *SOSR*, pages 148–161. ACM, 2021.

[40] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.

[41] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[42] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 29–42, 2013.

[43] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. ARROW: Restoration-Aware Traffic Engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 560–579, 2021.

[44] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[45] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 479–491, New York, NY, USA, 2015. Association for Computing Machinery.

[46] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A PROPERTIES OF HASH-BASED TREES

In this section, we provide an analysis on the impact on accuracy and detection speed for different hash-based tree parameters. Further, we provide formulas to compute collision probability and memory requirements for generic hash-based trees, depending on their width, depth and split.

A.1 Parameters analysis

Since hash paths identify entries affected by failures, the total number and length of hash paths influence the number of entries per counter. Both factors directly depend on the tree's depth and width, as the number of hash paths is equal to w^d, and their length is bounded by d. Increasing width and depth therefore increases accuracy, by making it more likely that leaf counters map to a single entry (see Appendix A.2 for details). Increasing depth and width, however, comes at the cost of higher memory occupation (see A.3 for details). Additionally, higher depths increase the number of counting sessions needed slowing down failure detection and making the failure harder to detect, thus decreasing accuracy. As such, width and depth regulate the tradeoff between accuracy on one side and memory and detection speed on the other. In Appendix D, we show this effect with a simulation with real traces.

In general, the detection speed for entries mapped to the tree depends on the performance of the zooming algorithm. As described in Section 4.2, the algorithm explores up to k^{d-1} paths in d counting sessions. Thus, the detection speed also depends on the split value k: higher split values speed up the detection of multi-entry failures (by a factor proportional to k), but it also requires more memory (i.e., to store a bigger tree).

The duration of counting sessions, which we also denote as *zooming speed*, affects detection speed, too: it is quite intuitive that shorter counting sessions tend to make detection faster. However, decreasing the zooming speed can also impact FANcY's accuracy, as it reduces the probability of observing packet losses during d consecutive counting sessions. In §5, figure 8 shows that a very small zooming speed might negatively affect detection accuracy (more traffic is needed).

A.2 Collision probability

Hash-based tree counters provide a scalable way of monitoring many traffic entries for reasonable amounts of memory. However, it comes at the price of possible collisions, which manifest as false positives. Therefore, faulty entries monitored by the hash-based tree can share hash path with other entries.

To know the probability of collisions in our system we can use Bloom filters with a single hash function theory [20]. The number of possible hash paths would be the size of our Bloom filter. We compute the number of hash paths (m) as: $m = w^d$. Since we only care about collisions on cells with faulty entries, the probability of a collision depends on the number of cells and faulty entries, as is computed below, where n is the number of faulty entries at a given time.

$$p = (1 - e^{-1/(\frac{m}{n})}) \tag{1}$$

The expected number of collisions (or false positives) depends on the number of different traffic entries (x) that cross the hash-based tree, and it can be computed as:

$$E(x) = p \cdot x \tag{2}$$

A.3 Tree nodes and memory

The amount of memory required for the hash-based tree depends on its width, split, depth, and whether it runs in pipelined mode. The number of tree nodes we need to save in memory can be computed as:

(1) With pipelining:

$$nodes(k, d) = \begin{cases} \frac{k^d - 1}{k-1} & \text{if } k > 1 \\ d & \text{otherwise} \end{cases} \tag{3}$$

(2) Without pipelining: $nodes(k, d) = k^{d-1}$.
(3) Without pipelining and split 1: $nodes(1, d) = 1$.

Finally, the total amount of memory needed for a hash-based tree (without counting state machine resources) with 32-bit counters, node width w, split k, and depth d is computed as: $2 \cdot 32 \cdot w \cdot nodes(k, d)$.

B FANCY IMPLEMENTATION IN AN INTEL TOFINO SWITCH

We now provide more details on our FANcY implementation and consumed resources.

B.1 Implementation

We first describe our implementation of the state machines and then we focus on how we support for hash-based trees.

State machines. While implementing each state is relatively simple (i.e., storing a state ID and possible counters in registers), supporting state transitions is not. In particular, we did not find a way to read a state, do relatively complex operations and then update the state in a single step. We therefore implement each state transition in two steps.

The first step only triggers the state transition and is based on a match-action table, called *next_state* table. This table defines all the possible state transitions. When a FANcY switch receives a packet, it reads the current state from a register and matches the packet against the *next_state* table, if a transition needs to be made, The switch logic will (i) write in the *state_lock* register, in order to avoid additional transitions while the state is being updated, (ii) store all the information needed to update the state in the current packet's metadata, and (iii) force the packet to cross the pipeline again to perform the state update.[3]

The second step actually performs the transition. The recirculated packet updates the stored state ID, resets the state counters (e.g., timers), and releases the *state_lock*. Based on prior and next states information in its metadata, the packet also triggers a

[3]For technical reason, we *resubmit* packets in ingress FSMs and *clone* packets in egress FSMs (making sure we remove unneeded clones)

transition-specific action: either drops the packet, performs a computation, or transforms it into a control message (*ACK*, *STOP*, etc.) to send out. A final note concerns time-based transitions (e.g., timers). Since time-based events are not supported by current switches, we approximate them using traffic and packet counts. In the absence of traffic, the internal traffic generator can be used.

Hash-based tree and zooming algorithm. We implement the hash-based tree (depth 3 and split 1) by using four register arrays. One register array, which we call *node register*, stores actual nodes of all the trees kept by the switch (i.e., one per port). The other three register arrays store metadata to support the zooming algorithm: for each tree, the *zooming stage* register array keeps information on the depth we are currently zooming in, the *max0* register indicates the counter at layer zero we are zooming in, and the *max1* register stores the same information but for the counter at layer one. The procedure to update counters in any tree T of width w is implemented as follows. Each incoming packet is hashed according to one hash function per tree's level. We then decide if the *node register* has to be updated by checking whether the *zooming stage* register for T is 0 (i.e., we always update counters when not zooming), or comparing the result of the packet's H_0 with *max0* (if the *zooming stage* register for T is 1) and packet's H_0, H_1 with *max0* and *max1* (if the *zooming stage* register for T is 2). If the *node register* has to be updated, we increase the counter at the address ($H_i \bmod w + o$), where i is the value stored in the *zooming stage* register for T and o is the port offset that identifies T within the *node register*.

In addition to increasing packet counters, we also support two other operations. First, the downstream switch sends to the upstream all T's counters in the *node register* at the end of each counting session. Since register register arrays can be accessed only once per packet, we recirculate packets w times to read all such counters from the *node register*.

Second, the upstream switch compares local counters in T with those reported from downstream switches. Again, since only one register can be read for each packet, we recirculate packets w times to compare the counters one by one. If the *zooming stage* register for T is 2, we simply report (to our reroute app or externally) all the counters with mismatching values. Otherwise, if the *zooming stage* register for T is 0 or 1, we need to compute the counter in T's *node register* with the biggest difference of values between upstream and downstream. We do so by storing the current maximum difference and counter index in a custom header of the packet that we recirculate. After all the counters in T's *node register* are compared, we finally copy the counter index in the recirculated packet's metadata to either *max0* or *max1* (depending on the current value in the *zooming stage* register) and increase the *zooming stage* register by one modulo three.

B.2 Hardware memory consumption

FANcY scales and fits very well in current hardware switches. Rhis section details the resources needed for each component in a 32-port Tofino switch. Note that the software and hardware implementations use the same data structures, however the hardware implementation runs non-pipelined hash-based trees, which heavily reduced the memory consumption. For more details on memory utilization for any type of hash-based tree see Sec A.3.

State machines. Each state machine uses three registers (at ingress and egress): State counter (or timer), current state, and state lock, 32, 8, and 8 bits, respectively. We need one array cell in each of those registers for each sub-state machine used by either dedicated counters or a hash-tree. For each state machine pair, FANcY needs $(32 + 8 + 8) \cdot 2 = 96$ bits. If we want to have 512 state machines per port in a 32-port switch, we need $96 \cdot 512 \cdot 32 = 192$ KB.

Dedicated counters. Each entry covered by dedicated counters requires one pair of 32-bit registers to count packets in each direction, $32 \cdot 2 = 64$ bits per entry per switch. Our implementation of FANcY includes 512 dedicated counters per port. The memory consumption of those counters in a 32-port switch is therefore $64 \cdot 512 \cdot 32 = 128$ KB.

Hash-based tree and zooming algorithm. Supporting any hash-based tree requires five registers in total. First, we need the two 32-bit registers where we will store tree's nodes. Then, at the egress pipe, we have three registers used by the zooming algorithm; zooming stage, max0 and max1, 8, 16, and 16 bits, respectively. Since we implement a hash-tree zooming algorithm without split and pipelining, we can reuse the same memory cells for each tree layer, considerably reducing the memory needed. The hash-based trees in our implementation have width $w = 190$. Each of them therefore needs $32 \cdot 2 \cdot 190 = 12160$ bits per port for the counters, and $8 + 16 + 16 = 40$ bits to keep zooming state. In total, for a 32-port switch we need $(12160 + 40) \cdot 32 = 47.6$ KB.

Rerouting. Supporting the rerouting logic also needs some switch memory. We use 3 registers (all at the ingress) for that, one for dedicated counter entries and one for failures detected with the hash-based tree. For dedicated counter entries, we use a 1-bit wide array, thus we need 1 bit per entry and port. For 512 entries and 32 ports, we need 2 KB. For failures detected via the hash-based tree, we additionally need to use a Bloom filter implemented as two 1-bit registers of 100K cells. The memory used for the rerouting is 26.4 KB.

Total memory. For a 32 port switch, with 512 dedicated counter entries, one hash-based tree of width 190 per port and depth 3 is 367.6 KB (394 KB with rerouting).

C CAIDA TRACES USED IN FANCY EVALUATION

Table 5 lists the real-world traces in our evaluation, detailing some of their characteristics.

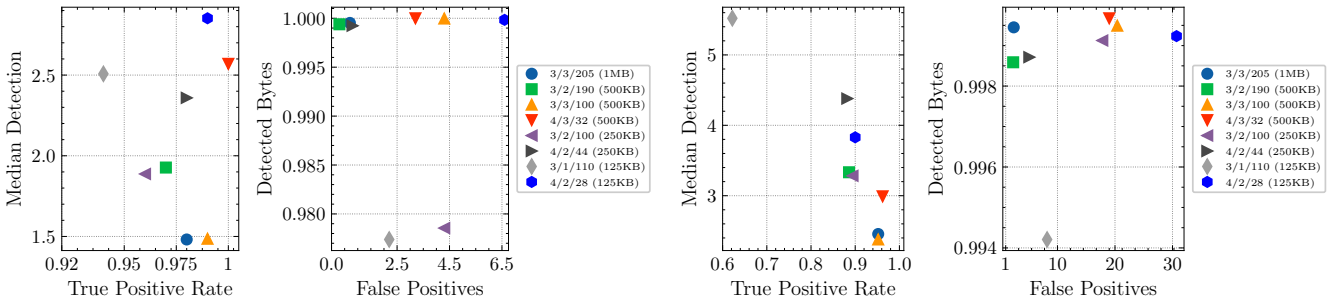
D SENSITIVITY ANALYSIS OF FANCY'S PARAMETERS

In this section, we perform experiments to show the impact of changing the values of tree's parameters. For that, we compare a set of system designs using Internet traces.

Methodology. We compare different FANcY hash-based tree configurations by running it using the trace with most prefixes ($\approx 560K$,

Trace ID	Link	Date	Bit Rate	Packet rate	Flow rate	Trace Size	Duration
1	caida-equinix-chicago.dirB	19-06-2014	6.25 Gbps	759.1 Kpps	28.3 Kfps	163 GB	3719 s
2	caida-equinix-nyc.dirA	19-04-2018	3.86 Gbps	557 Kpps	26.4 Kfps	125 GB	3719 s
3	caida-equinix-nyc.dirB	16-08-2018	5.79 Gbps	2.03 Mpps	104.5 Kfps	465 GB	3719 s
4	caida-equinix-nyc.dirB	17-01-2019	4.72 Gbps	1.56 Mpps	90.7 Kfps	345 GB	3720 s
Total						1.1 TB	4.1 h

Table 5: List of CAIDA traces [21] that we use to evaluate FANcY.



(a) Speed vs TPR & Bytes vs FP for 10 failures

(b) Speed vs TPR & Bytes vs FP for 50 failures

Figure 11: Left: comparison of eight different hash-based tree performances when 10 failures happen at the same time. Right: same with a burst of 50 failures. Hash-based trees can detect the vast majority of affected prefixes (specially the ones carrying most of the traffic) at scale. Memory usage can be considerably reduced at the price of detection speed and false positives. The size of the failure burst has a negative impact on all the metrics. Trees with a bigger split perform better in such cases.

ID 4 C) such that we can see how does that impact the number of false positives. During the simulation, we fail (100% loss) either 10 or 50 prefixes at the same time. This process is averaged over 10 runs with different randomly selected prefixes. Note that we only fail prefixes that can be detected at the zooming speed and depth used by the system under test($\approx 120k$ prefixes). To make the comparison insightful, we selected designs with different memory sizes (from 125 KB to 1 MB). We use the pipelined version of FANcY, thus, we need to reserve memory space for all nodes in the tree. Systems are defined as follows: $depth/split/width(Memory)$, as you can see in Figure 11 legend. For this experiment, we do not use dedicated counter entries, thus we allocate 100% of the memory to the hash-based tree.

Split increases TPR and reduces detection speed. Designs with big split have the best true positive rates and lowest median detection speeds for failures affecting many entries. Figure 11b, left side, shows that the fastest and more accurate designs have a split of 3. The gray design, with a split of 1, has the worst detection speed and TPR.

Depth increases detection time with a slight decrese in TPR. Designs with a depth of 4 have the biggest detection times, which is expected since they require more zooming stages. We can also see, that although it has an impact, increasing the depth does not drastically decrease the TPR.

Memory can be traded by speed without sacrificing too much TPR. We can find relatively cheap designs like 4/2/44, which have a decent good TPR, and small FP being one of the cheapest designs. However, it has some of the worst median detection times.