Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach, Edgar Costa Molero, and Maria Apostolaki, ***ETH Zurich;***
Alberto Dainotti, ***CAIDA/UC San Diego;*** **Stefano Vissicchio,** ***UC London;***
Laurent Vanbever, ***ETH Zurich***

https://www.usenix.org/conference/nsdi19/presentation/holterbach

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach,* Edgar Costa Molero,* Maria Apostolaki*
Alberto Dainotti,† Stefano Vissicchio,‡ Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

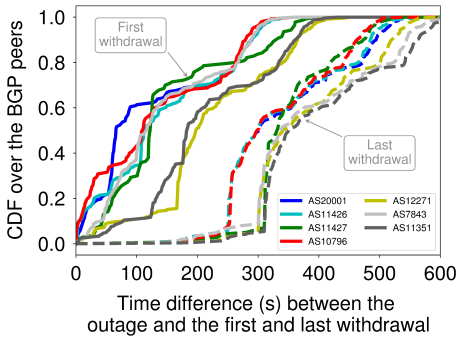


Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

In short, a fundamental question is still open: *Is it possible to build a fast-reroute framework for ISPs that can converge in $O(seconds)$ for both local and remote failures?*

Blink: *fast, data-driven* convergence upon remote failures. We answer this question affirmatively by developing Blink, a *data-driven* fast-reroute framework built on top of programmable data planes. Blink key insight is to reroute based on data-plane signals rather than control-plane ones. Quickly after a failure, data-plane traffic indeed exhibits a predictable behavior: all the TCP endpoints start retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. With Blink, we show that this signal can be efficiently tracked at line rate and enables sub-second convergence after most remote failures.

Key challenges. Tracking failure signals in the data plane is challenging for at least three reasons. First, monitoring all flows is impossible because of memory constraints. At the same time, randomly sampling flows often results in tracking useless flows, *e.g.,* ones that seldom transmit. We address this problem by developing a *flow selector* which automatically evicts inactive flows and replaces them with active ones.

Second, packet loss routinely happens in the Internet, *e.g.,* due to temporary congestion. Rerouting upon any retransmission would result in huge, and counterproductive traffic shifts. We address this challenge by: (i) focusing on timeout-induced retransmissions, which are infrequent (as we confirmed analyzing real traces); and (ii) leveraging the fact that failures affect many flows simultaneously.

Third, data-plane signals provide no information about the root cause of the problem.Worse, uncoordinated rerouting decisions can lead to forwarding issues such as blackholes, forwarding loops, and oscillations. In Blink, we solve these problems by also making the backup selection data-driven, *i.e.,* by tracking if flows resume after rerouting them.

We fully implemented Blink in $P4_{16}$. Our evaluation, which includes experiments on a real Barefoot Tofino switch, shows that Blink retrieves connectivity within 1 s for the vast majority of the considered failure cases.

Main contributions. Our main contributions are:

- A new approach for quickly recovering connectivity upon remote failures based on data-plane signals (§2).
- The Blink pipeline, which enables programmable data planes to track failure signals at line rate and to automatically retrieve connectivity (§3 and §4).
- An implementation[1] of Blink in Python and $P4_{16}$ (§5).
- An evaluation of Blink using synthetic and real packet traces, emulations, and hardware experiments (§6).
- A discussion on how to deploy Blink, along with how to protect it from malicious and crafted traffic (§7).

[1] Our source code is available at: `https://blink.ethz.ch`

2 Key Principles and Challenges

In this section, we first show that TCP traffic exhibits a characteristic pattern upon failures (§2.1). We then discuss the key challenges and requirements to detect such a pattern, and recover connectivity by rerouting the affected prefixes, while operating entirely in the data plane, at line rate (§2.2).

2.1 Data-plane signals upon failures

Consider an Internet path (A,B,C,D) carrying tens of thousands of TCP flows, destined to thousand prefixes, in which the link (B,C) suddenly fails. We are interested in monitoring the data-plane "failure signal" perceived at A, with the goal of enabling A to detect it and to also recover connectivity by rerouting traffic through a different path (if any). Observe that A is not adjacent to the failure, *i.e.,* the failure is remote.

As the link (B,C) fails, the TCP endpoints stop receiving acknowledgements (ACKs), and each of them will timeout after its retransmission timeout (RTO) expires, which will cause it to reset its congestion window to one segment and start retransmitting the first unacknowledged segment. Since the RTO is computed according to the RTT observed, each TCP endpoint will retransmit at a different time. Specifically, each TCP endpoint adjusts its RTO using the following relation: RTO = sRTT + 4*RTTVAR (see [31]), where sRTT corresponds to the smoothed RTT, and RTTVAR corresponds to the RTT variation. After each retransmission, each TCP endpoint further doubles its RTO (exponential backoff).

We illustrate the behavior of a TCP flow experiencing a failure in Figure 2. We assume that the TCP endpoint has an estimated RTO of 200 ms and that its congestion window can hold 4 packets. We denote by t the time at which the TCP endpoint transmits the first packet following the failure. The TCP endpoint experiences consecutive RTO expirations and retransmits the packet with sequence number 1000 at time $t+200ms$, $t+600ms$, $t+1400ms$, etc. We experimentally verified that this behavior is similar across all TCP flavors implemented in the latest Linux kernel.

When multiple flows experience the same failure, the signal obtained by counting the overall retransmissions consists of "retransmission waves". Since this behavior is systematic, pronounced, and quick, we leverage it in Blink to perform failure detection in the data plane. This suggests Blink does not depend on specific TCP implementation details and would keep working effectively with future congestion control algorithms as long as they exhibit a similar behavior upon failures.

Note however the shape of these retransmission waves, *i.e.,* their amplitude and width, depends on the distribution of the estimated RTTs. As an illustration, Figure 3 shows the retransmission count for a trace that we generated with the ns-3 simulator [3] after simulating a link failure (according to the methodology in §6.1). In the left diagram, we used the distribution of the average RTTs of the TCP flows from an actual traffic trace (#8 in Table 3 in §E). In the right diagram, we increased the RTTs of this distribution by 1.5 to obtain

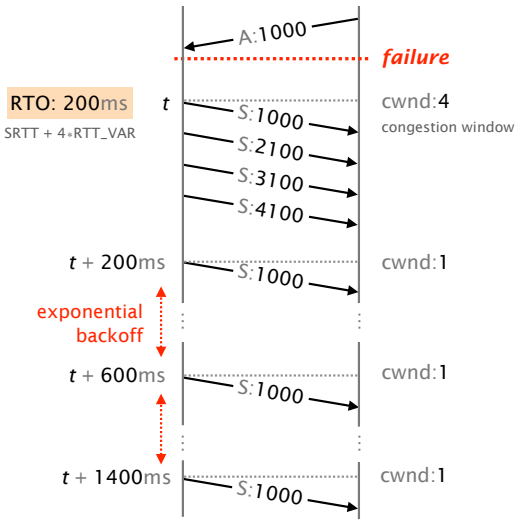


Figure 2: After the failure, a TCP flow keeps retransmitting the last unacknowledged segment according to an exponential backoff. The exact timing of the retransmissions depends on the estimated RTO before the failure (here 200ms).

a larger standard deviation. We can clearly see the waves of retransmissions appearing within a second after each failure. RTT distributions with small variance make the flows more synchronized they will be when retransmitting. This translates into narrow peaks of retransmissions with a high amplitude. Conversely, if the flows have very different RTTs (*i.e.,* the variance is high), the peaks will have a smaller amplitude and will spread over a longer time. We elaborate on the challenges deriving from these observations hereafter.

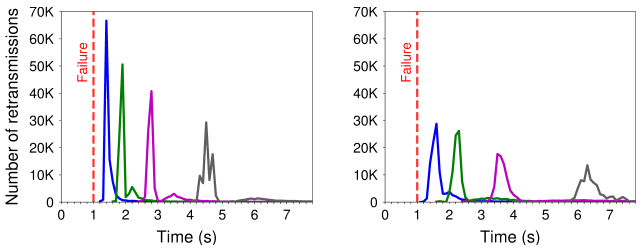
2.2 Key challenges and requirements when fast rerouting using data-plane signals

We now highlight four key challenges and requirements that must be addressed to: (i) efficiently capture the failure signal we just described; and (ii) recover connectivity. We describe in §3 how does Blink address them entirely in the data plane.

Dealing with noisy signal. To discover its fair share of bandwidth, a TCP endpoint keeps increasing its transmission rate until a packet loss is detected, triggering a retransmission. TCP retransmissions therefore occur naturally, even without network failures. Likewise, minor temporary congestion events can also lead to bursts of packet drops, which will trigger subsequent bursts of retransmissions, again, without necessarily implying a failure.

Requirement 1: A data-plane-driven fast-reroute system should only react to major disruptive events while being immune to noise and ordinary protocol behavior.

Dealing with fading signals. As shown in Figure 3, the amplitude of the signal (*i.e.,* the count of TCP retransmissions)



(a) Failure Signal (RTT 1×) (b) Failure Signal (RTT 1.5×)

Figure 3: The signal generated by TCP flows experiencing a connectivity problem is characteristic and composed of subsequent waves of retransmissions (in different colors). The waves have decreasing amplitude and increasing width.

quickly fades with the backoff round as the compounded signal spreads over longer and longer periods.

Requirement 2: A data-plane-driven fast-reroute system should catch the failure signal within the first retransmission rounds.

Mitigating the effect of sampling. As tracking retransmissions in real-time requires state, monitoring *all* flows is not possible. As such, a fast-reroute system will necessarily have to track and detect failures using a subset of the flows. Yet, not all flows are equally useful when it comes to failure reaction: intuitively, highly active flows will retransmit almost immediately, while long-lived flows might not retransmit at all (if no packet was sent recently). From a fast-reroute viewpoint, tracking non-active flows is useless.

Requirement 3: A data-plane-driven fast-reroute system should select the flows it tracks according to their activity.

Ensuring forwarding correctness without control plane. While data-plane signals are faster to propagate than control-plane ones, they carry no information about the cause of the failure and how to avoid it. As such, simply rerouting to a backup next-hop upon detecting a problem might not work, as it might also be affected by the failure. Worse, the problem can even be at the destination itself, in which case no alternative next-hop will actually work. Given this lack of precise information, a data-plane-driven fast-reroute system has no other choice but trying and observing.

Requirement 4: A data-plane-driven fast-reroute system should select its backup next-hops in a data-driven manner, verifying that traffic resumes.

3 Overview

In this section, we provide a high-level description of Blink. We first focus on its data-plane implementation (§3.1) and how it: (i) selects flows to track; (ii) detects failures; and (iii) reroutes traffic. We then describe how Blink can be deployed at the network level (§3.2).

3.1 Blink, at the node level

Figure 4 describes the overall workflow of a Blink data-plane pipeline. The pipeline is essentially composed of three consecutive stages: (i) a *selection* stage which efficiently identifies active flows to monitor; (ii) a *detection* stage which analyzes RTO-induced retransmissions across the monitored flows and looks for any significant increase; and (iii) a *rerouting* stage which is in charge of retrieving connectivity by probing alternative next-hops upon failure. We now briefly describe the key ingredients behind each stage and provide details in §4.

Selecting flows to track. For efficiency and scalability, a Blink node cannot track all possible 700k+ IP prefixes or even all the flows destined to some prefixes. An initial design choice thus concerns which prefixes to track, and which specific flows to track for the selected prefixes.

Any approach based on data-plane signals is able to effectively monitor only the prefixes carrying a certain amount of packets. Blink is no exception, and therefore focuses on the most popular destination prefixes. While this might seem a limitation, it is actually a feature: the Internet traffic is typically skewed [32], and a very limited fraction of prefixes carries most of the traffic, while the rest of the prefixes see little to none. Blink can thus reroute the vast majority of the traffic by tracking a limited number of prefixes. We designed Blink to accommodate at least 10 k prefixes in current programmable switches (§5).

Regarding which flows to track for a prefix, Blink adopts a simple but effective strategy. For each monitored prefix, the Flow Selector tracks a very small subset (64, by default) of *active flows—i.e.,* flows that send at least one packet within a moving time window (2 s by default). Tracked flows are replaced as soon as they become inactive, or after a given timeout (8.5 min by default) even if they remain active. We did not reuse heavy hitter detection algorithms such as [36], since they are designed to offer higher accuracy than we need (heaviest flows instead of just active ones) at the expense of additional complexity and resources.

Detecting failures. A central idea of our approach is to infer remote failures, affecting a destination prefix, from the loss of connectivity for a statistically significant number of previously active flows towards that destination. While possible in principle, Blink does not look at the flows progression to detect a failure, as only a subset of the flows may be affected, *e.g.,* because of load-balancing. This enables Blink to detect partial failures (see §D).

The detection stage looks for evidence of connectivity disruption across the flows identified by the Flow Selector. It stores key information on the last seen packet for each flow and determines if a new packet traversing the data-plane pipeline is a duplicate of the last seen one – an hallmark of RTO-induced TCP retransmissions (see Figure 2). Based on this check, for each destination prefix, it monitors the number of flows with at least one recent retransmission over a sliding time window of limited size (800 ms, by default). When the majority of the monitored flows experience at least one retransmission in the same time window, Blink infers a failure.

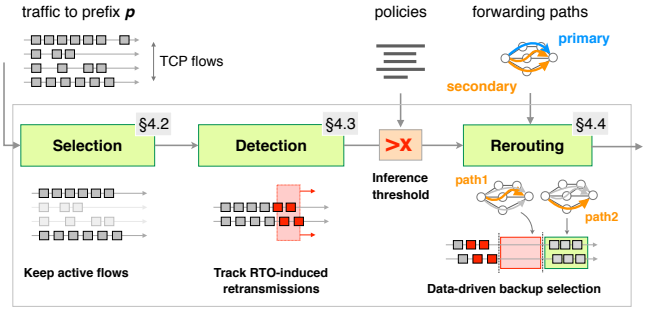


Figure 4: Blink data-plane workflow and key ingredients.

Rerouting quickly. When Blink detects a failure, the Rerouting Module quickly reroutes traffic by modifying the next-hop to which packets are forwarded, at line rate. In Blink's current implementation, the decisions of both when to reroute and to which backup next-hop to reroute are configurable by the operator based on their policies, as we believe that operators want to be in charge of this critical, network-specific operation.

When rerouting, the Rerouting Module sends few flows to each backup path to check which one is able to restore connectivity. It then uses the best and working one for all the traffic. The next-hops are configured at runtime by the operator to re-align the data-plane forwarding to the control-plane (*e.g.,* BGP) routes when the control plane has converged.

3.2 Blink, at the network level

The "textbook" deployment of Blink consists in deploying it on all the border routers of the ISP to track all the transit traffic. In this deployment, border routers either reroute traffic locally (if possible) or direct it to another border router (*e.g.,* through an MPLS tunnel). Of course, nothing prevents the deployment of Blink inside the ISP as well. In fact, Blink also works for intra-ISP failures, *e.g.,* local to the Blink node or on the path from the Blink switch and an ISP egress point.

Blink is partially deployable. Deploying Blink on a single node already enables to speed up connectivity recovery for all traffic traversing that particular node. Also, Blink requires no coordination with other devices: each Blink node autonomously extracts data-plane signals from the traversing packets, infers major connectivity disruptions, and fast reroutes accordingly. To avoid forwarding issues, Blink veri-

fies the recovery of connectivity for the rerouted packets by monitoring the data plane (see §4.4.2).

When rerouting, Blink also notifies the control plane, and possibly the ISP operator. This enables coordination with the control plane (*e.g.,* future SDN controllers), such as imposing the next-hop upon control-plane convergence, or discarding routes that are not working in the data plane.

4 Data-plane design

In this section, we describe the data-plane pipeline that runs on a Blink node, its internal algorithms, design choices and parameter values (that we further discuss in §A). Figure 5 depicts the four main components of the Blink data-plane pipeline: the Prefix Filter (§4.1), the Flow Selector (§4.2), the Failure Inference (§4.3), and the Rerouting Module (§4.4).

4.1 Monitoring the most important prefixes

To limit the resources used by Blink, the operator should activate Blink only for a set of important prefixes. A sensible approach would be to activate it for the most popular destination prefixes, as they carry most traffic, although nothing prevents the operator to select other prefixes – as long as there is enough TCP traffic destined to each of them (§6.1.1). To activate Blink for a prefix, the control plane adds an entry in the metadata table at runtime which matches the traffic destined to this prefix using a longest prefix match. Traffic destined to a prefix for which Blink is not active goes directly to the last stage of the data-plane pipeline and is forwarded normally (*i.e.,* find the next-hop and replace the layer 2 header).

The metadata table attaches to the matched packets a distinct ID according to their destination prefix. As memory (*e.g.,* register arrays) is often shared between the prefixes, this ID is used as an index to the memory. Observe that Blink could combine prefixes with common attributes (*e.g.,* origin AS or AS path) and which are likely to fail at the same time by mapping them to the same ID. This would increase the intensity of the signal, and would allow Blink to cover more traffic. Additionally, packets that do not carry useful information, *i.e.,* non-TCP traffic, and certain signaling-only packets such as SYN and ACK packets with no payload are not considered by Blink and are directly sent to the final stage.

4.2 Selecting active flows to monitor

Packets destined to a monitored prefix go to the Flow Selector, which will select a limited number of active flows (64 per prefix), and keep information about each of them.

Limiting the number of selected flows. Each flow for a given prefix is mapped to one of the 64 cells of a per-prefix flow array using a 6-bit hash of the 4-tuple [2]. While we expect many flows to collide in the same cell, only one occupies a cell. This is enforced by storing the flow_key, namely a 32-bit hash of the 4-tuple in each cell of the flow array.

[2]The 4-tuple includes source and destination IP and the port numbers.

Flows colliding in the same cell are possible candidates to substitute the flow currently occupying that cell when it becomes inactive. It can happen that two flows mapped to the same cell have the same flow_key, in which case both would end up occupying the same cell, causing Blink to mix packets from two distinct flows and thus preventing it to correctly detect retransmissions for either flows. However, since we use a total of 38 bits (6 bits to identify the cell and 32 bits for the flow_key) to identify each flow, such collisions will rarely happen. The probability of collision can be computed from a generalization of the *birthday problem*: given n flows, what is the probability that none of them returns the same 38 bit hash value? This probability is equal to $\frac{2^{38}!}{(2^{38}-n)!} * \frac{1}{2^{38n}} \approx e^{-\frac{n(n-1)}{2*2^{38}}}$. With $n = 10,000$ flows for a given prefix, the probability to have a collision is only 0.02%.

Replacing inactive flows. The challenge behind selecting active flows is that flows have different packet rates, which also change over time, *e.g.,* an active flow at time t may not be active anymore at time $t + 1s$. A naive *first-seen, first-selected* strategy would clearly not work, because the selected flows might send packets at such a low rate that they would not provide any timely information upon a connectivity disruption – simply because there is no packet to retransmit[3].

The Flow Selector monitors the activity of each selected flow by tracking the timestamp of the last packet seen in the register last_pkt_ts. As soon as the difference between the current timestamp and last_pkt_ts is greater than an eviction timeout, the flow is evicted and immediately replaced by another flow colliding in the same cell. A TCP FIN packet also causes immediate eviction. Intuitively, flow eviction makes the Flow Selector work very well for prefixes which have many high-rate flows at any moment in time, or a decent fraction of long-living ones – which we expect to be often the case for traffic towards popular destinations. Our evaluation on real traffic patterns (see §6) confirms that this simple strategy is sufficient to quickly infer major connectivity disruptions.

Calibrating the eviction timeout. A remaining question for this component of the pipeline is how to dimension the eviction timeout. On one hand, we would like to evict flows as soon as their current packet rate is not amongst the highest for that prefix. On the other hand though, Blink needs to keep track of the flows long enough to see the first few packet retransmissions induced by a RTO expiration upon connectivity interruptions. Indeed, an eviction timeout of few hundred milliseconds is likely to be too low in many cases, since a flow takes *at least* 200ms to issue the first pair of duplicate packets[4] (see §2). By default, the eviction timeout is set to 2 s , which ensures to detect up to two pairs of consecutive duplicates for typical TCP implementations.

[3]Our experimental evaluation in §6 confirms this intuition

[4]200ms only happens if there is no new packet between the first unacknowledged packet and the first retransmission. Also, remember that Blink considers only consecutive duplicates as packet retransmissions.

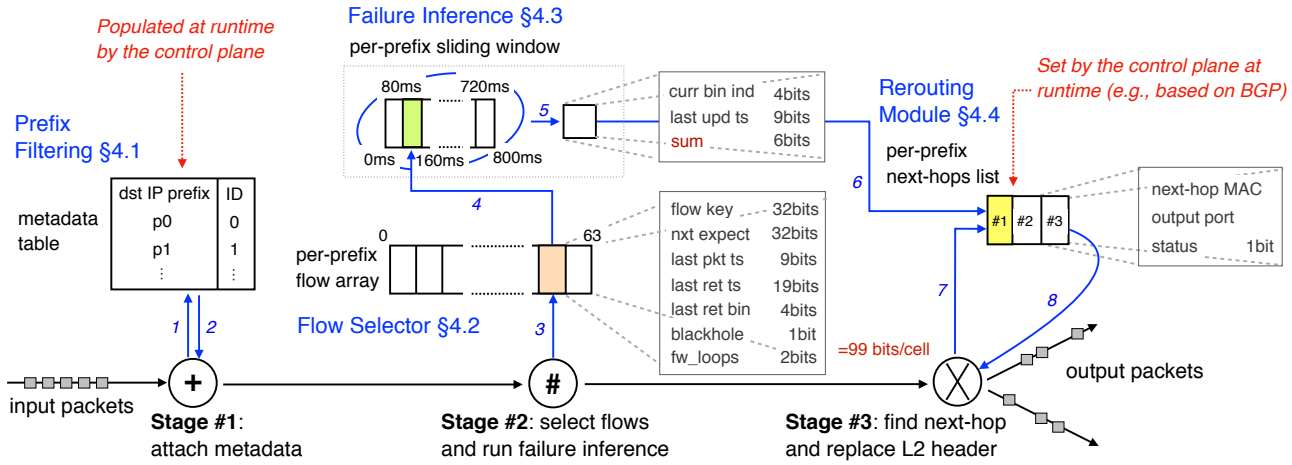


Figure 5: Blink data-plane pipeline and its data structures. The blue arrows indicate all the steps a packet goes through.

4.3 Detecting failures

We now describe how Blink detects RTO-induced retransmissions on the set of selected flows, and uses this information to accurately infer failures.

Detecting RTO-induced retransmissions. A partial or full retransmission of payload of the TCP packet can be detected by comparing, the sum of its sequence number and payload length to the corresponding sum of the previous packet of the same flow. For example, in Figure 2 when the packet S:1000 (packet with sequence number 1000) arrives Blink will store 2100 (sum of sequence number and payload). If the next arriving packet triggers storing 2100 as well, a retransmission is detected. Observe that we store the expected sequence number per flow instead of the current to account for cases where a packet is only partially acked.

Our design targets consecutive retransmissions for two reasons. First, RTO-induced retransmissions are consecutive (see Figure 2), whereas congestion-induced retransmissions (*i.e.,* noise) are likely to be interleaved by non-retransmissions, and hence will (correctly) not be detected. Second, this detection mechanism requires a fixed number of memory per flow, regardless the flow's packet rate.

Counting the number of flows experiencing retransmissions over time. Figure 3 shows that the TCP signal upon a failure is short and fading over time. To quickly and accurately detect the compounded signal across multiple flows, we use a per-prefix sliding window. To implement a sliding window of size k seconds in P4, we divide it in 10 consecutive bins of 6-bit each, each storing the number of selected flows experiencing retransmissions during $k/10$ seconds. As a result, instead of sliding for every packet received, the sliding window moves every $k/10$ seconds period. More bins can improve the precision but would require more memory. This design enables us to implement the sliding window in P4 using only three information per prefix: (i) current_index, the index of the bin focusing on the current period of time, (ii) sum, the sum of all the 10 bins, and (iii) last_ts_sliding, the timestamp in millisecond precision[5] of the last time the window slid. The additional 19-bit and 4-bit per-flow information last_ret_ts and last_ret_bin are also required to ensure that a flow is counted maximum one time during a time window. We provide more details about the implementation in §B.

Calibrating the sliding window. The duration of the sliding window affects the failure detection mechanism. A long time window (*e.g.,* spanning several seconds) has more chance to include unrelated retransmissions (*e.g.,* caused by congestions), whereas a short time window (*e.g.,* 100ms) may miss a large portion of the retransmissions induced by the same failure because of the different RTO timers. We set the duration of the sliding window to 800ms, with 10 bins of 80ms. First, because the minimum RTO is 200ms, a 800ms sliding window ensures to include all the retransmissions induced by the failure within the first second after the failure. Second, because under realistic conditions (in terms of RTT [5, 18, 34] and RTT variation [5]), flows would often send their first two retransmissions within the first second after the failure.

Inferring failures. A naive strategy consisting in inferring a failure when *all* the selected flows experience retransmissions would result in a high number of false negatives due to the fact that some flows may not send traffic during the failure, or simply because some flows have a very high RTT (*e.g.,* >1s). On the other hand, inferring a failure when only few flows experience retransmissions may result in many false positives because of the noise. As a result, by default Blink infers a failure for a prefix if the majority of the monitored flows (*i.e.,* 32) destined to that prefix experience retransmissions.

[5] We explain in §B how Blink can obtain millisecond precision

4.4 Rerouting at line rate

As soon as Blink detects a failure for a prefix, it immediately reroutes the traffic destined to it, at line rate. We first show in §4.4.1 how Blink maintains the per-prefix next-hops list used for (re)routing traffic. Then, we show in §4.4.2 how Blink avoids forwarding issues when it reroutes traffic.

4.4.1 Maintaining the per-prefix next-hops list

To reroute at line rate, Blink relies on pre-computed per-prefix backup next-hops. The control plane computes the next-hops consistently with BGP routes and specific policies defined by the operator. For each prefix, Blink maintains a list of next-hops, which are sorted according to their preference (see Figure 2). Each next-hop has a status bit. To reroute at line rate, Blink deactivates the primary next-hop by setting its status bit to 1 (*i.e.,* not working). Per-prefix next-hops are stored in register arrays and are updated at runtime by the controller, *e.g.,* whenever a new BGP route is learned or withdrawn. If a next-hop is not directly connected to the Blink node, Blink can translate it into a forwarding next-hop using IGP (or MPLS) information, as a normal router would do.

Falling back to the primary next-hop after rerouting. After an outage, BGP eventually converges and Blink updates the primary next-hop and use it for routing traffic. However, Blink cannot know when BGP has fully converged. Our current implementation waits for a fixed time (*e.g.,* few minutes, so that BGP is likely to have converged) after rerouting before falling back to the new primary next-hop. We acknowledge that this approach might not be optimal (*e.g.,* it potentially sacrifices path optimality), but it guarantees packet delivery by using policy-compatible routes and avoids possible disruptions caused by BGP path exploration [28]. Investigating a better interaction with the control plane is left for future work.

4.4.2 Avoiding forwarding issues

Since Blink runs entirely in the data plane, it likely reroutes traffic before receiving any control-plane information possibly triggered by the disruption. In addition, even when carefully selecting backup next-hops (*e.g.,* by taking the most disjoint AS path with respect to the primary path), we fundamentally cannot have a-priori information about where the root cause of a future disruption is, or where the backup next-hop sends the rerouted traffic after the disruption. As a result, Blink *fundamentally cannot prevent* forwarding issues such as blackholes (*i.e.,* when the next-hop is not able to deliver traffic to the destination) or forwarding loops to happen. The good news, though, is that Blink includes mechanisms to *quickly react to forwarding issues* that may inevitably occur upon rerouting.

Probing the backup next-hops to detect anomalies. When rerouting, Blink reacts to forwarding anomalies by probing each backup next-hop with a fraction of the selected flows in order to assess whether they are working or not. For example, with 2 backup next-hops, one half of the selected flows is rerouted to each of them. The non-selected flows destined to this prefix are rerouted to the preferred and working backup next-hop. Blink does this in the data plane using the per-prefix next-hops list.

When a backup next-hop is assessed as not working, Blink updates its status bit. After a fixed period of time since rerouting (1s, by default), Blink stops probing the backup paths and uses the preferred and working one for all the traffic, including the selected flows. If all the backup next-hops are assessed as not working, Blink reroutes to the primary next-hop and falls back to waiting for the control plane to converge.

Avoiding blackholes. Blink detects blackholes by looking at the proportion of restarted flows. After rerouting, Blink tags a flow as restarted by switching its blackhole bit to 1 as soon as it sees a packet for this flow which is not a retransmission. When the probing period is over, Blink assesses a backup next-hop as not working if less than half of the flows routed to that next-hop have restarted. The duration of the probing period (1s) is motivated by our goal of restoring connectivity at a second-level time scale, while also providing retransmissions with a reasonable time for reaching the destination through the backup next-hop and triggering the restart of the flows. For example, if Blink reroutes 778 ms after a failure (the median case, see §6.1.1) and assuming a reasonable RTT (*e.g.,* the median case in [5, 18, 34]), it is likely that the rerouted flows will send a retransmission and receive the acknowledgment (if the next-hop is working) within the following 1 s period.

Breaking forwarding loops. Blink detects forwarding loops by counting the number of duplicate packets for each flow. The key intuition is that forwarding loops have a quite strong signature: the same packets are seen over and over again by the same devices. This signature is very similar to the TCP signature upon a failure, where TCP traffic sources start resending duplicate copies of the same packets for every affected flows, at increasingly spaced epochs. As a result, the algorithm used by Blink to detect retransmissions also detects looping packets. To differentiate between normal retransmissions and looping packets, Blink relies on the delay between each duplicate packet. TCP can send for a flow up to 2 retransmissions in 1 s because of the exponential backoff (see §2.1), whereas a packet trapped in a forwarding loop can be seen many more times by the Blink node. Hence, Blink counts the number of duplicate packets it detects for each flow after the rerouting using the information fw_loops stored in each cell of the flow array, and tags a backup next-hop as not working by switching its status bit as soon as it detects more than 3 duplicate packets for a flow rerouted to this backup next-hop.

Observe that this mechanism reacts very quickly to the most dangerous loops, *i.e.,* the ones that recirculate packets very fast and hence are most likely to overload network links and devices. Longer and slower loops are mitigated in at most 1 s, as Blink assumes the respective next-hop cannot deliver packets to the destination (*i.e.,* there is a blackhole).

5 Implementation

We have fully implemented the data-plane pipeline of Blink as described in §4 in ≈900 lines of $P4_{16}$ [37] code and in Python. We have also developed a $P4_{Tofino}$ implementation of Blink that runs on a Barefoot Tofino switch [8]. Our $P4_{Tofino}$ implementation currently only supports two next-hops, one primary and one backup. Unlike our $P4_{16}$ implementation, our $P4_{Tofino}$ implementation uses the resubmission primitive[6] in two cases: whenever the Flow Selector evicts a flow, or if two retransmissions from same flow are reported within 800ms, *i.e.,* the duration of the sliding window. Note that these two cases only occur for the set of selected flows (*i.e.,* only 64 flows per prefix, see §4.2).

Our implementations of Blink only require one entry in the metadata table, as well 6418 bits of memory (*i.e.,* registers) for each prefix monitored. This number is fixed, *i.e.,* only this amount of memory is required regardless of the amount of traffic. This is an important feature for a system such as Blink, which is intended to run on hardware with strong limitations. On current programmable switches, we expect Blink to support at least 10k prefixes. We explain in §C how we precisely derive the resources required by Blink.

6 Evaluation

We evaluate Blink's accuracy, speed, and effectiveness in selecting a working next-hop based on simulations and synthetic data (§ 6.1, § 6.2). We then evaluate Blink using real traces and actual hardware (§ 6.3).

6.1 Blink's failure detection algorithm

Packet traces of real Internet traffic are hard to gather, and for the few traces publicly available [9, 12], there is no ground truth about possible remote failures on which Blink should reroute. Still, it makes little sense to evaluate Blink on traffic with non-realistic characteristics, or without knowing if Blink is correctly or incorrectly rerouting packets. We therefore adopt the following evaluation methodology.

Methodology. We consider 15 publicly available traces [9,12] (listed in §E), accounting for a total of 15.8 hrs of traffic and 1.5 TB of data. For each prefix, we extract the distributions of flow size, duration, average packet size, and RTT.[7] We then run simulations with ns-3 [3] on a dumbbell topology similar to [38], where traffic sources generate flows exhibiting the same distribution of parameters than the one extracted from the real traces.

In some of our simulations, we introduce a failure after 10 seconds on the single link connecting the sources with the destinations, thus affecting all flows. We refer the reader to §D for an evaluation on partial failures. In other experiments, we introduce random packet drops and no failure at all. We collect traffic traces for all simulations, feed them to our Python-based implementation of Blink one by one, and check if and when our system would fast reroute traffic.

Baselines. Since we are not aware of any previous work on real-time failures detection on the basis of TCP-generated signals, we compare Blink against two baseline strategies. The first strategy, *All flows*, consists in monitoring up to 10k flows for each prefix, and rerouting if any 32 of them sees retransmissions within the same time window. This strategy provides an upper bound on Blink's ability to reroute upon actual failures while ignoring memory constraints. The second strategy, ∞ *Timeout*, is a variant of Blink where flows are only evicted when they terminate (with a FIN packet), and never because of the eviction timeout. This strategy assesses the effectiveness of Blink's flow eviction policy.

6.1.1 Blink often detects actual failures, quickly

We first evaluate Blink's ability to detect connectivity disruptions. For each real trace in our dataset, we randomly consider 30 prefixes which see a large number of flows ($>$ 1000 flows in the trace), and we generate 5 synthetic traces per prefix, each with a different number of flows starting every second (from 100 to 500 flows generated *per second*) and each containing a failure at a preconfigured moment in time.

We then compute the True Positive Rate (TPR) of Blink on these traces. For each synthetic trace, we check whether Blink detects the failure (True Positive or TP) or not (False Negative or FN). The TPR is computed over all the tested synthetic traces, and is equal to $TP/(TP+FN)$.

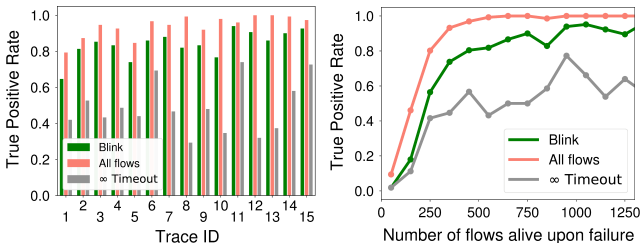
Figure 6a shows the TPR of Blink and our baseline strategies as a function of the real trace used to generate the synthetic ones. As expected, the *All flows* strategy exhibits the best TPR among the three considered strategies at the cost of impractical memory usage. We see that Blink has a TPR which is very close to (*i.e.,* less than 10% lower than) the *All flows* strategy—while tracking only 64 flows. Overall, Blink correctly reroutes more than 80% of the times for 13 traces out of 15, with a minimum at 65% and a peak at 94%. At the other extreme, the TPR of the ∞ *Timeout* strategy is much lower than Blink, below 50% for most traces, highlighting the importance of Blink's flow eviction policy.

As the RTT of the flows affects the failure signal used by Blink to detect failures (see §2.1), we also look at the TPR as a function of the RTT. On the synthetic traces with a median RTT below 50 ms (resp. above 300 ms), Blink has a TPR of 90.6% (resp. 76.0%). This shows that Blink is useful even when flows have a high RTT.

As a follow-up, we then analyzed how much Blink's TPR varies with the number of flows active upon the failure (an important factor for Blink's performance). Figure 6b shows that Blink's TPR unsurprisingly increases if there are more flows active during the failure. With very few active flows, Blink cannot perform well, since the data-plane signal is too

[6] A resubmitted packet goes twice in the ingress to take more actions while being forwarded by the switch.

[7] To measure the RTT of the flows, we use the time difference between the SYN and ACK packets sent by the initiator of a connection as described in [22, 34].



(a) Different traffic patterns (b) Different number of flows

Figure 6: Blink has high TPR when relatively few flows (*e.g.,* more than 350) are active upon the failure.

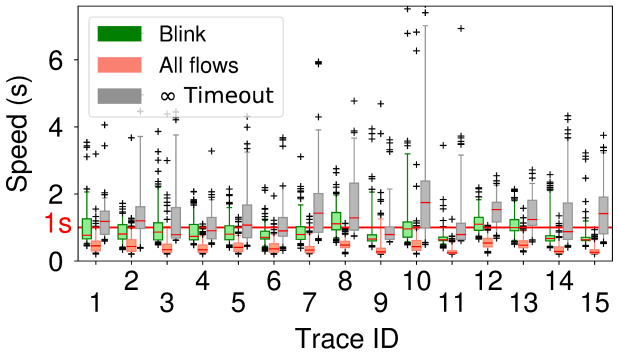


Figure 7: Blink is fast, for all traffic patterns.

weak. However, Blink's TPR is already around 74% when about 350 flows are active, and reaches high values (more than 90%) with about 750 active flows. Again, Blink is much closer to the *All flows* strategy than to the ∞ *Timeout* one, although the *All flows* strategy reaches higher levels of TPR for lower number of active flows. These results suggest that Blink is likely to have a high TPR in a real deployment since we expect to see ≫750 active flows for popular destinations.

Not only does Blink detect failures in most cases, but it also recovers connectivity quickly upon failure detection. Figure 7 shows the time needed for Blink to restore connectivity for each of the real traces used to generate the synthetic ones, restricting on the cases where Blink detects the failure. Each box shows the inter-quartile range of Blink's reaction time. The line in each box depicts the median value; the whiskers show the 5th and 95th percentile. Blink retrieves connectivity in less than 778 ms for 50% of the traces, and within 1 s for 69% of the traces. The *All flows* strategy restores connectivity within 365 ms in the median case, whereas the ∞ *Timeout* strategy needs 1.07 s (median). Naturally, Blink is faster when the RTT of the flows is low. On the synthetic traces with a median RTT below 50 ms, Blink reroutes within 625 ms in the median case. Yet, when the median RTT is above 300 ms, Blink is still fast and reroutes within 1.2 s in the median case.

6.1.2 Blink distinguishes failures from noise

One may wonder if Blink's ability to detect failures may not be due to it overestimating disruptions. By design, Blink cannot detect a failure without TCP retransmissions. Hence, the question is if Blink tends to overreact to relatively few, unrelated retransmissions, *e.g.,* induced by random packet loss.

To verify this, we generate synthetic traces with no failure but with an increasing level of random packet loss (from 1% to 9%) for all traffic. The trace synthesis follows our methodology of mimicking characteristics of real traffic for one prefix. For each real trace and loss percentage, we repeat the trace generation for 10 randomly extracted prefixes which see a large number of flows. For this experiment, we generate traces from 1-minute simulations where many (*i.e.,* 500) new flows start every second to ensure that Blink's Flow Selector is filled with flows, all potentially sending retransmissions.

For all these synthetic traces, we check whether Blink detects a failure (FP) or not (TN) and compute the False Positive Rate (FPR) as $FP/(FP+TN)$. Contrary to what happens for the TPR in §6.1.1, we expect *All flows* to be a worst case scenario as it sees all the retransmissions across all flows. On the other hand, ∞ *Timeout* should perform better than Blink because inactive flows (which are not evicted) do not contribute to the number of observed retransmissions.

Table 1 shows the FPR as a function of packet loss. Below 4% packet loss, Blink never detects failures. Between 4% and 7%, Blink incorrectly detected a failure for *one* synthetic trace out of the 150 generated. This indicates that Blink would work well under realistic traffic (we confirm this in §6.3.1), where the packet loss is often below these values. As a reference, the *All flows* strategy has an extremely high FPR, around 60% (resp. 85%) for traces with 1% (resp. 2%) packet loss. The ∞ *Timeout* strategy has only one false positive when the packet loss is between 5% and 10%, which, rather than a feature, is an artifact of tracking non-active flows.

Summary. Our results show that Blink strikes a good balance between detection of actual failures and robustness to noise (*i.e.,* TCP retransmissions not originated from a prefix-wide connectivity disruption). Blink's tradeoff is much better than the naive strategies: not evicting flows would significantly lower Blink's ability to correctly reroute upon failures, while monitoring all crossing flows comes with high sensitivity to noise (in addition to a likely impractical memory cost). Blink also recovers connectivity quickly (often within 1 s in our experiments) when it detects a disruption.

6.2 Blink's rerouting algorithm

We now focus on the Blink's Rerouting Module. Our design ensures that rerouting is done entirely in the data plane, at line rate —we confirm this by experimenting with a Tofino switch, as described in §6.3. In this section, we therefore evaluate whether Blink is effective in rerouting to a working next-hop.

Methodology. We emulate the network shown in Figure 8 in a virtual machine attached to 12 cores (2.2 GHz). The P4 switch has three possible next-hops to reach the destination, R1 being

packet loss %	1	2	3	4	5	6	7	8	9
				False Positive Rate (%)					
Blink	0	0	0	0.67	0.67	0.67	0.67	1.3	2.0
All flows	59	85	93	94	95	96	97	97	98
∞ timeout	0	0	0	0	0.67	0.67	0.67	0.67	0.67

Table 1: Blink avoids incorrectly inferring failures when packet loss is below 4%.

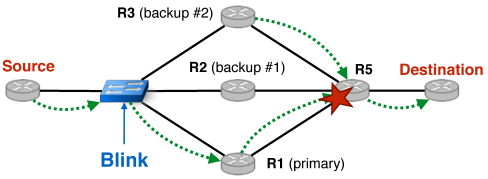


Figure 8: The network used to evaluate Blink's rerouting. Arrows indicate forwarding next-hops (R2 uses different next-hops depending on the experiment).

the primary next-hop, R2 the most preferred backup next-hop and R3 the less preferred backup one. R1 and R3 use R5 as next-hop to reach the destination. R2 uses a different next-hop to reach the destination depending on the experiment, thus we do not depict it in the figure.

We emulate the P4 switch by running our $P4_{16}$ implementation of Blink in the P4 behavioral model software switch [4]. The P4 switch running Blink is linked to a Mininet network emulating the other switches. The source and the destination are Mininet hosts running TCP cubic.

We start 1000 TCP flows from the source towards the destination. To show the effectiveness of the Flow Selector, 900 flows have a low packet rate (chosen uniformly at random between 0.2 and 1 packet/s) while only 100 have a high packet rate (chosen uniformly at random between 2.5 and 20 packet/s). We use `tc` to control the per-flow RTT (chosen uniformly at random between 10 ms and 300 ms), and to drop 1% of the packets on the link between R5 and the destination in order to add a moderate level of noise. We first start the 900 flows with a low packet rate, so that the Flow Selector first selects them. Right after, we start the 100 remaining flows. Finally, after 20 s to 30 s, we fail the link between R1 and R5.

Blink quickly detects and breaks loops. We configure R2 to use the P4 switch as next-hop to reach the destination so that it creates a forwarding loop (by sending traffic back to the source) when Blink reroutes traffic to R2. Figure 9a shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Prior the failure, the traffic goes through R1, the primary next-hop. Upon the failure, Blink probes if any of the available next-hops can recover connectivity: it sends half of the flows in the Flow Selector to R2 and the other half to R3. All the remaining flows go to R2 (preferred over R3). Blink detects the forwarding loop induced by R2 very quickly (only 8 packets were captured on R2) and immediately deactivates this next-hop to reroute all the traffic to R3, restoring connectivity within a total of 800 ms.

Blink quickly detects and routes around blackholes. In a separate experiment, we configure R2 to use R5 as next-hop, and we fail the link between R2 and R5 in addition to the one between R1 and R5. Figure 9b shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Upon the failure, Blink reroutes to R3 half of the selected flows, and to R2 the other half of the selected flows plus all the non-selected ones (since R2 is preferred over R3). However, because the link between R2 and R5 is down, the packets sent to R2 are just dropped by R2. After 1 s, Blink detects the blackhole and reroutes all the traffic to R3, restoring connectivity. The total downtime induced by the failure is 1.7 s.

6.3 Blink in the real world

So far, we have evaluated Blink with simulations and emulations. We now report on experiments that we run on real traffic traces and on a Barefoot Tofino switch.

6.3.1 Running Blink on real traces

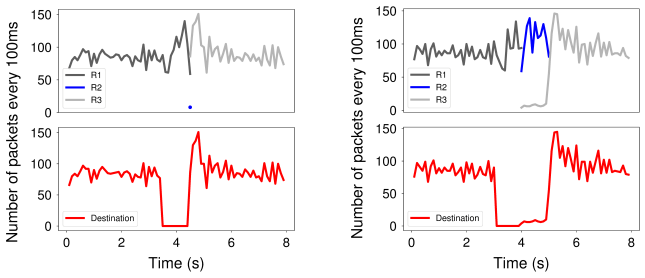
In §6.1, we use simulated (but realistic) traffic traces to gain some confidence on Blink's accuracy in detecting connectivity disruptions. An objection might be that our synthetic traces are not fully realistic. We therefore run Blink on the original real traces listed in §E and tracked when it detects a failure[8]. Observe that unlike in §6.1, here we do not simulate failures. Since we do not have ground truth about actual failures in real traces, we manually checked each case for which Blink detected a failure so as to confirm the connectivity disruption.

Over the 15.8 hrs of real traces, Blink detected 6 failures. In these 6 cases, the retransmitting flows represent 42%, 57%, 71%, 82%, 82% and 85% of all the flows active at that time and destined to the affected prefix. These numbers confirms that Blink is *not* sensitive to normal congestion events, and only reroutes in cases where a large fraction of flows experience retransmissions at the same time.

6.3.2 Deploying Blink on Barefoot Tofino switches

We finally evaluate our $P4_{Tofino}$ implementation of Blink on a Barefoot Tofino Wedge 100BF-32X. To do so, we generate TCP traffic between two servers connected via our Tofino switch running Blink. The server receiving the traffic has a primary and a backup physical link with the Tofino switch. We generate 1000 flows, 900 of which have a low packet rate and 100 a high one (similarly to §6.2). To show the influence of the RTTs on Blink when running on Tofino, we run two experiments, one with sub-1ms RTT, and another one in which we use `tc` to simulate for each flow an RTT chosen uniformly at random between 10 ms and 300 ms. After 30 s, we simulate

[8] We omitted failures detected for 73 prefixes (out of 2.28M) which *constantly* showed high-level of retransmissions (>20% of the flows retransmitting >50% of the time). Blink could detect such outliers at runtime.



(a) Blink quickly breaks loops (b) Blink reacts to blackholes

Figure 9: Traffic measurements quantifying Blink's speed in reacting to forwarding anomalies upon rerouting.

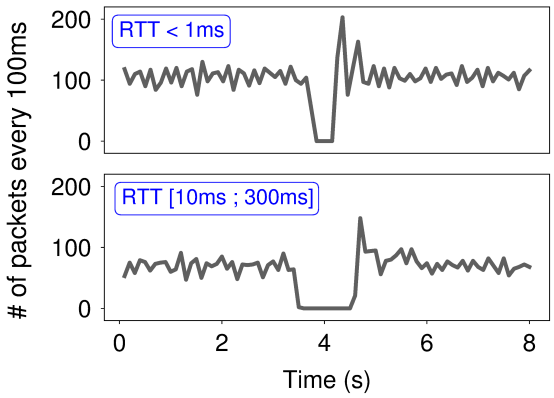


Figure 10: A Tofino switch running Blink retrieves connectivity within 460 ms if the RTT of the affected flows is below 1 ms (top figure); or within 1.1 s if the per-flow RTT ranges between 10 ms and 300 ms (bottom figure).

a failure on the primary path, and measure the time Blink takes to retrieve connectivity via the backup link.

Blink-Tofino managed to restore connectivity in <1s. Blink retrieves connectivity in only 460ms with sub-1ms RTT, and in about 1.1 s when the RTT of the flows is between 10 ms and 300 ms. We obtain comparable results (470 ms of downtime with sub-1 ms RTT) when running the same experiments with 3,000 flows among which 300 has a high packet rate.

7 Deployment considerations

We now discuss three possible operational concerns when deploying Blink in a real ISP network: security, adaptability and interaction with deployments of Blink in other ISPs.

Security. As potentially any Internet user can generate data-plane traffic, security could be a concern for running Blink in operational networks. The main threat is that malicious users could manipulate Blink to reroute traffic by sending fake retransmissions for flows towards a victim destination.

Blink's design itself significantly mitigates security risks. For any given destination, Blink reroutes traffic if most of the 64 monitored flows retransmit nearly at the same time. The monitored flows are selected among all active flows for the given destination, and flows sending more packets are intuitively privileged by the Flow Selector substitution policy. Hence, a brute-force attack would have to generate an amount of traffic comparable to the legitimate traffic destined to the attacked prefix in order to have a reasonable success chance. The fact that Blink focuses on popular destinations, typically attracting large traffic volumes and many flows, implies that the attacker would have to generate lots of traffic to trick Blink– a condition under which the attack would be quite visible, could be monitored and potentially mitigated at runtime.

A Blink-savvy attacker could instead produce few flows with a high and constant packet rate, and which never terminate, so that when one of them is selected by the Flow Selector, it remains selected forever. Blink will eventually monitor 32 of them, making the attacker ready to operate. However, Blink is built to evict a flow, *even if active*, after a fixed time (8.5 min by default, see §C). This implies that the attacker only has a short time window during which she can perform the attack. Evicting active flows more frequently (*e.g.,* every few seconds instead of 8.5 min) would better prevent such attacks, but would affect the failure detection mechanism of Blink because some retransmissions can be missed.

While the above mechanisms do not make Blink bullet-proof, we believe that they make our system's attack surface reasonably small. We plan to perform a deeper analysis of Blink's security concerns in future work.

Adaptability. Clearly, a challenge in a real deployment of Blink is how to set its parameters correctly (see §A). This is hard because operators have different requirements, and traffic can exhibit varying characteristics. In §6, we show that it is definitely possible to set the parameter values so that Blink works well in real situations. Yet, in a real deployment, we envision that Blink could first be run in "learning mode", where it sends notifications to the controller instead of rerouting traffic. The controller then evaluates the accuracy of the system, for instance using control-plane data, and turns Blink on if the accuracy is good, or tune some parameters otherwise.

Internet scale deployment. So far, we have described Blink's deployment in a single network (see §3). Of course, all ISPs have the same incentives to deploy Blink (*i.e.,* for fast connectivity recovery), so we envision that multiple, possibly all, ISPs might deploy Blink. Multi-AS deployment of Blink makes rerouting trickier. For example, if an Internet path traverses multiple Blink switches, it is not clear which ones will reroute, and whether the resulting backup path will be optimal. Blink switches can also interfere with each other. For example, if a Blink switch reroutes traffic to a backup path, a downstream Blink switch in the original path may lose part of the data-plane signal, preventing it to detect the failure. Finally, Blink's rerouting can also increase the likelihood of creating inter-domain loops, since Blink selects backup next-

hops based on BGP information, which might not be truthful if the downstream switches also run Blink.

While a full characterization of Blink's behavior in an Internet-scale deployment is outside the scope of this paper, Blink's design already guarantees some basic correctness properties. Blink already monitors for possible forwarding loops, and quickly breaks them by using additional backup next-hops (see §4.4.2). After having explored all possible backup next-hops, Blink also falls back to the primary next-hop indicated by the control-plane, even if not working: this would prevent oscillations where two or more Blink switches keep changing their respective next-hops in the attempt to restore connectivity. Finally, Blink switches do use BGP next-hops after BGP convergence.

Path optimality is much harder to guarantee within a system like Blink, where network nodes independently reroute traffic, without any coordination. However, we believe that path optimality can be transiently sacrificed [9] in the interest of restoring Internet connectivity as quickly as possible.

8 Related Work

We now discuss related work beyond fast reroute frameworks for local failures (such as [14, 29, 35]), which we already discussed in the introductory section.

Recent research explores how to quickly localize remote inter-domain failures and reroute upon them, assuming no data-plane programmability. Prominently, ARROW [30] uses tunnels between endpoints and ISPs in combination with special control-plane packets. Gummadi et al. [17] infer failures from the data plane, and attempt to recover connectivity by routing indirectly through a small set of intermediaries. Several other approaches (*e.g.,* [13, 19, 21]) infer connectivity problems from BGP messages. Some of them also diagnose these problems and fast-reroute upon their detection. By inferring failures from data-plane packets, Blink is fundamentally faster than control-plane based solutions (by minutes, *e.g.,* in Figure 1). Also, unlike [17,30], Blink is deployable on a single node, and does not require interaction with other devices.

Data-Driven Connectivity [25] (DDC) ensures connectivity *within a network* via data-plane mechanisms. Chiesa et al. [11] consider generalizations of the DDC approach, and study the relationship between the resilience achieved through data-plane primitives and network connectivity. The work from Sedar et al. [33] shows how to program a hardware switch so that it automatically reroutes the traffic to a working path upon failure of directly connected links. In contrast to the above line of work, Blink fast recovers upon failures occurring in other networks, and uses data-plane programmability to detect connectivity disruptions rather than to dynamically configure post-failure paths.

Data-plane traffic has also been widely used in the past for ex-post measurement analyses. For example, WIND [20] infers network performance problems, including outages, from traffic traces by leveraging (among others) structural characteristics of TCP flows. In Blink, we perform online packet analysis, at line rate, but only to infer major connectivity disruptions – with simple yet effective algorithms that fit the limited resources of real switches.

Few approaches monitor traffic using a programmable data plane. DAPPER is an in-network solution using TCP-based signals to identify the cause of misbehaving flows (whether the problem is in the network or not) [16]. Blink does not aim at identifying the cause of a particular flow failing but rather that many flows (for the same prefix) fail at the same time. In addition, unlike Blink, DAPPER requires symmetric routing for its analysis, which is often not the case in ISP environments. Sivaraman et al. [36] propose a heavy-hitter detection mechanism running entirely in the data plane. As Blink, it stores flows in an array and relies on flow eviction to keep track of the heaviest ones. Unlike [36], Blink looks for active flows instead of the heaviest ones, on a per-prefix basis.

Recent work from Molero et al. [26] shows that key control-plane tasks such as failure detection, notifications and new path computations can be offloaded to the data plane. Such systems could directly benefit from Blink, *e.g.,* by leveraging it to detect remote failures and trigger network-wide convergence accordingly.

9 Conclusions

Blink is the first data-driven fast-reroute framework targeting remote connectivity failures. By operating entirely in the data plane, at line rate, Blink restores connectivity in $O(s)$. We evaluate Blink with realistic traces, taking into account different traffic conditions as well as noise due to significant packet loss. Our results show that Blink enables sub-second connectivity retrieval in the majority of the scenarios, while preventing unnecessary traffic shifts in the presence of noise. By deploying Blink on a Barefoot Tofino switch, we also confirm that it can run in commercial programmable devices.

10 Acknowledgments

We are grateful to the NSDI anonymous reviewers and our shepherd, Harsha Madhyastha, for their insightful comments. We also thank the members of the Networked Systems Group at ETH Zurich for their valuable feedback. This work was supported by a Swiss National Science Foundation Grant (Data-Driven Internet Routing, #200021-175525). This research was also supported by the U.S. Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via contract number 70RSAT18CB0000015.

[9] Even with an Internet-wide deployment of Blink, path optimality will be restored when the control plane converges to post-failure paths, or operators will manually solve connectivity disruptions after Blink's notifications.

References

[1] CNN - Time Warner Cable comes back from nationwide Internet outage, 2014. https://money.cnn.com/2014/08/27/media/time-warner-cable-outage/index.html.

[2] RIPE RIS Raw Data, 2016. https://www.ripe.net/data-tools/stats/ris/.

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

[4] P4 behavioral model., 2018. https://github.com/p4lang/behavioral-model.

[5] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in tcp round-trip times. In *ACM IMC'03*, New York, NY, USA.

[6] Alia K. Atlas and Gagan Choudhury and David Ward. IP Fast Reroute Overview and Things we are struggling to solve. https://bgp.nu/~dward/IPFRR/IPFRR_overview_NANOG.pdf.

[7] A. Atlas and A. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, September 2008.

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

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

[10] CAIDA. The UCSD Network Telescope. https://www.caida.org/projects/network_telescope/.

[11] Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. On the resiliency of static forwarding tables. *IEEE/ACM Transactions on Networking*, 2017.

[12] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the wide project. In *USENIX ATEC'00*.

[13] Anja Feldmann, Olaf Maennel, Z Morley Mao, Arthur Berger, and Bruce Maggs. Locating Internet routing instabilities. *ACM SIGCOMM CCR*, 2004.

[14] Clarence Filsfils. BGP Convergence in much less than a second, 2007. Presentation NANOG 23.

[15] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *ACM SIGCOMM CCR*, 2005.

[16] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *ACM SOSR'17*, 2017.

[17] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the reliability of internet paths with one-hop source routing. In *USENIX OSDI'04*.

[18] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *ACM CoNEXT '16*.

[19] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. SWIFT: Predictive Fast Reroute. In *ACM SIGCOMM'17*.

[20] Polly Huang, Anja Feldmann, and Walter Willinger. A non-instrusive, wavelet-based approach to detecting network performance problems. In *ACM SIGCOMM Workshop on Internet Measurement, 2001*.

[21] Umar Javed, Italo Cunha, David Choffnes, Ethan Katz-Bassett, Thomas Anderson, and Arvind Krishnamurthy. PoiRoot: Investigating the Root Cause of Interdomain Path Changes. In *ACM SIGCOMM, 2013*.

[22] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *ACM SIGCOMM CCR*, 2002.

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

[24] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. *ACM SIGCOMM CCR*, 2000.

[25] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *USENIX NSDI'13*.

[26] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Hardware-accelerated network control planes. In *ACM HotNets '18*.

[27] University of Oregon. Route Views Project, 2016. www.routeviews.org/.

[28] Ricardo Oliveira, Beichuan Zhang, Dan Pei, Rafit Izhak-Ratzin, and Lixia Zhang. Quantifying path exploration in the internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 269–282. ACM, 2006.

[29] P. Pan, G. Swallow, and A. Atlas. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, May 2005.

[30] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. *ACM SIGCOMM CCR*, 2014.

[31] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[32] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM CCR*, 2012.

[33] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. Supporting emerging applications with low-latency failover in p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, NEAT '18. ACM, 2018.

[34] S. Shakkottai, N. Brownlee, A. Broido, and k. claffy. The RTT distribution of TCP flows on the Internet and its impact on TCP based flow control. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), Mar 2004.

[35] M. Shand and S. Bryant. IP Fast Reroute Framework. RFC 5714, January 2010.

[36] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR'17*.

[37] The P4 Language Consortium. P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[38] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. *ACM SIGCOMM CCR*, 2006.

Appendix

A Blink parameters

In this section, we list in Table 2 the main parameters used by Blink, and show how each of them can affect the performance of the system. We denote as TPR the True Positive Rate over all the synthetic traces generated from the 15 real traces listed in Table 3 and used in §6.1.1. The TPR with Blink's default values over all the synthetic traces is 83.9%. FPR denotes the False Positive Rate and is computed similarly as in §6.1.2.

B Implementation of a sliding window in P4$_{16}$

In this appendix section, we show how we implemented a sliding window in the P4$_{16}$ language.

Blink uses one sliding window per prefix to count the number of flows experiencing retransmissions over time among the selected flows (§4.3). Besides the 10 bins, Blink needs three other meta-information: (i) current_index, the index of the bin focusing on the current period of time, (ii) sum, the sum of all the 10 bins and (iii) last_ts_sliding, the timestamp in millisecond precision of the last time the window slid. When Blink detects a retransmission, it increments by one both the value associated with the bin at the index current_index and the sum. Upon reception of a packet at timestamp t, and assuming the window covers a period of k millisecond, if $t - \text{last\_ts\_sliding} > k/10$, the window slides by one doing the following operations. To find the index of the bin that has expired, Blink computes $(\text{current\_index} + 1) \bmod 10$[10]. Then, it subtracts to sum the value stored in the expired bin, and then resets it. Then, Blink makes current_index point to the expired bin and finally updates last_ts_sliding to last_ts_sliding $+ k$. As a result, the counter sum always returns the number of flows experiencing retransmissions during the last $9/10k$ to k seconds.

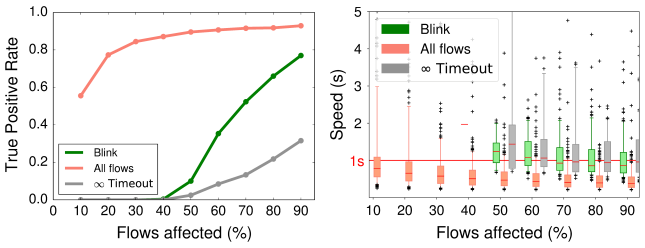
It can happen that a flow sends several retransmissions within a time window. To avoid summing several retransmissions from same flow within the same time window, Blink uses two additional per-flow metadata called last_ret_ts and last_ret_bin. The former stores the timestamp of the last retransmission reported for the corresponding flow. The later stores the bin index corresponding to this timestamp. Consider that a retransmission for a flow is reported at time t, then if $t - \text{last\_ret\_ts} < k$, Blink decrements by one the value in the bin at the index last_ret_bin, and increments by one the value associated to the current bin. The sum remains the same, and last_ret_ts is set to the current timestamp and last_ret_bin is set to the current bin index.

C Hardware Resource Usage

Blink is intended to run on programmable switches with limited resources. As a result, we designed Blink to scale based on the number prefixes it monitors, and not on the actual amount of traffic destined to those prefixes. In this section, we derive the resources required by Blink to work for one prefix, and show that it can easily scale to thousands of prefixes.

First, for every prefix, Blink needs one entry in the metadata table. Then, for each selected flows, the Flow Selector needs 99 bits (see Figure 5). As Blink monitors 64 flows per prefix, a total of $64 * 99 = 6336$ bits are required for one prefix. To save memory, Blink does not store the timestamps (*e.g.,* last_pkt_ts and last_ret_ts) in 48 bits (the original size of the metadata), but instead approximates them using only 9 bits for seconds and 19 bits for milliseconds. To obtain the second (resp. millisecond) approximation of the current timestamp, Blink shifts the original 48-bit timestamp to the right. To fill in 9 (resp. 19) bits, Blink also resets the timestamps every 512 s ($\approx$ 8.5min) by subtracting to the original 48-bit timestamp a reference_timestamp. The reference_timestamp is simply a copy of the original 48-bit timestamp (stored in a register shared by all the prefixes) that is updated only every 512 s. Note that the Flow Selector evicts a flow if the current timestamp is lower than the timestamp of the last packet seen for that flow, which happens whenever the timestamps are reset (*i.e.,* every $\approx$ 8.5min). This is actually good for security, as we explain in §7.

The sliding window requires 10 bins of 6 bits each, as well as $4 + 9 + 6 = 19$ bits to store additional information (see Figure 5), making a total of 79 bits. For the rerouting, Blink only requires 1 status bit per next-hop. With three next-hops, 3 extra bits are required. In total, for one prefix, Blink requires $6336 + 79 + 3 = 6418$ bits. As current programmable switches have few megabytes of memory, we expect Blink to support up to 10k prefixes, possibly even more.



(a) Accuracy (b) Speed

Figure 11: Blink is accurate and fast even for partial failures affecting 70% or more of the flows.

D Evaluating Blink on partial failures

In this section, we evaluate the performance of Blink upon partial failures, and compare it to our two baselines, the *All flows* and ∞ *Timeout* strategies. We consider partial failures to be those affecting only a portion of the traffic (*e.g.,* due to load-balancing).

For this evaluation, we randomly picked 10 prefixes from

[10] As the modulo operator is not available in P4$_{16}$, we implement this with if-else conditions.

Component	Name	Default value	Tradeoff
Flow Selector §4.2	Eviction timeout	2s	With a short eviction time (*e.g.,* 0.5s) flows can be evicted while they are retransmitting, reducing the TPR to 66.3%. With a longer eviction time (*e.g.,* 3s) inactive flows take more time to be substituted by active ones, reducing the TPR to 77.7%
	Number of cells per prefix	64	Monitoring a small fraction of flows may result in a FPR increase. For example, with only 16 cells, the FPR is 2% for only a 3% of packet loss. However, the bigger the number of cells the smaller the amount of prefixes we can monitor due to memory constrains. With 64 cells (=64 flows monitored per prefix) Blink can support at least 10k prefixes.
Failure Inference §4.3	Sliding window duration	800ms	A long time window (*e.g.,* 1.6s) is more likely to report all the retransmitting flows, increasing the TPR to 89.8%, but also reports more unrelated retransmissions, increasing the FPR (0.67% for 3% of packet loss). A shorter time window (*e.g.,* 400ms) limits the FPR (0% for 9% of packet loss) but decreases the TPR to 49.4%.
	Sliding window number of bins	10	More bins increases the precision, at the price of using slightly more of memory. 10 bins give a precision $> 90\%$.
	Inference threshold	50%	A lower threshold, such as 25% (*i.e.,* 16 flows retransmitting when using 64 cells) gives a better TPR (94.8%), but increases the FPR to 7.3% for 4% of packet loss.
Rerouting Module §4.4	Backup next-hop probing time	1s	A longer probing period better prevents wrongly assessing a next-hop as not working, at the price of waiting more time to reroute.

Table 2: Parameters used by Blink, with their default values, and how they can affect the performance of the system.

each real trace listed in Table 3, and generated 1 synthetic trace for each of them, following the guidelines described in §6.1. For each trace, we simulated partial failures with 9 different intensities (from 10% to 90% of the flows being affected). For these synthetic traces, 1223 flows (resp. 264) were active upon the failures in the median case (resp. 10th percentile).

Blink works in the majority of the cases for failures affecting 70% or more of the flows. Figure 11a shows the TPR of Blink as a function of the percentage of flows affected by the failure. Unsurprisingly, because Blink needs to detect at least 32 flows experiencing retransmissions to detect the failure, the TPR is close to 0% if the failure affects less than 50% of the flows. For failures affecting 70% of the flows (resp. 90%), Blink works for 53% (resp. 77%) of the failures. The *All flows* strategy performs well even for small failures affecting 20% of traffic, whereas the ∞ *Timeout* performs badly even for failures affecting 90% of the traffic.

Blink restores connectivity within one second in the median case for failures affecting at least 70% of the flows. Figure 11b shows the time needed for Blink to restore connectivity upon a partial failure. Logically, as we decrease the amount of affected flows, the detection speed of Blink increases. Yet, Blink is able to restore connectivity within 1 s for the majority of the cases if a failure affects 70% of the flows or more.

E Real traces used in the evaluation

In order to evaluate Blink with different traffic patterns (see §6), we use 15 real traces from different years, captured on different links and provided by different organizations, namely CAIDA [9] and MAWI [12]. Table 3 lists them and some of their characteristics.

Trace ID	Name	Date	Duration	Bit Rate	Trace Size	RTT median
1	caida-equinix-chicago.dirA	29-05-2013	3719 s	1631 Mbps	67 GB	200.22 ms
2	caida-equinix-chicago.dirB	29-05-2013	3719 s	2119 Mbps	73 GB	155.65 ms
3	caida-equinix-sanjose.dirA	21-03-2013	3719 s	2920 Mbps	122 GB	104.94 ms
4	caida-equinix-sanjose.dirB	21-03-2013	3719 s	1618 Mbps	82 GB	191.11 ms
5	caida-equinix-chicago.dirA	19-06-2014	3719 s	1629 Mbps	60 GB	209.72 ms
6	caida-equinix-chicago.dirB	19-06-2014	3719 s	6271 Mbps	163 GB	160.91 ms
7	caida-equinix-sanjose.dirA	19-06-2014	3719 s	3722 Mbps	144 GB	169.71 ms
8	caida-equinix-chicago.dirA	17-12-2015	3776 s	2540 Mbps	111 GB	240.18 ms
9	caida-equinix-chicago.dirB	17-12-2015	3776 s	3151 Mbps	99 GB	68.30 ms
10	caida-equinix-chicago.dirA	21-01-2016	3819 s	2250 Mbps	126 GB	224.09 ms
11	caida-equinix-chicago.dirB	21-01-2016	3819 s	4959 Mbps	143 GB	69.57 ms
12	caida-equinix-nyc.dirA	15-03-2018	3719 s	3027 Mbps	94 GB	306.76 ms
13	caida-equinix-nyc.dirA	19-04-2018	3719 s	3893 Mbps	125 GB	283.82 ms
14	mawi-samplepoint-F	12-04-2017	7199 s	878Mbps	74 GB	124.14 ms
15	mawi-samplepoint-F	07-05-2018	900 s	1098 Mbps	10 GB	156.20 ms
Total			15.8 h		1.5 TB	

Table 3: List of 15 real traces that we use to evaluate Blink. Altogether, they cover a total of 15.8 hrs of traffic.