ABSTRACT
Packet loss due to link corruption is a major problem in large warehouse-scale datacenters. The current state-of-the-art approach of disabling corrupting links is not adequate because, in practice, we often cannot disable all the corrupting links without violating capacity constraints. In this paper, we show that, it is feasible to implement link-local retransmission at sub-RTT timescales to completely mask corruption packet losses from the transport endpoints. Our system, Protego, employs a range of techniques to (i) keep the packet buffer requirement low, (ii) recover from tail packet losses without employing timeouts, and (iii) preserve packet ordering. We implement Protego on the Intel Tofino switch and show that for a 100G link with a loss rate of $10^{-2}$, Protego can reduce the loss rate by up to 6 orders of magnitude while incurring only an 8% reduction in effective link speed. By detecting and eliminating tail packet losses, Protego improves the 99.9th percentile flow completion time (FCT) for TCP and RDMA by 51x and 66x respectively. Finally, we also show that in the context of datacenter networks, it is not always necessary to preserve packet ordering when performing link-local retransmissions, and that simple out-of-order retransmission is often sufficient to significantly mitigate the impact of corruption packet loss for short TCP flows.

1 INTRODUCTION
Optical links are commonly used as switch-to-switch links in modern datacenter networks [59]. Unfortunately, external factors such as physical damage, bending, or contamination due to airborne dirt particles, can cause optical attenuation and make optical links susceptible to data transmission errors [16, 59]. As a result, packet losses due to corruption on optical links in large warehouse-scale datacenters are common. Alibaba’s recent study of hundreds of real-world service tickets showed that about 18% of the packet drops that caused network performance anomalies (NPAs) were due to packet corruption [57]. Another large-scale study across 15 Microsoft datacenters consisting of 350K optical links showed that the number of packets lost due to corruption is comparable to those lost due to congestion [59].

At the same time, Ethernet link speeds continue to increase, having increased from 25G [27] in 2016 to 400G [30] in recent years. This increase has been achieved through a combination of using multiple parallel PHY lanes, higher baudrate, and denser modulation. Figure 1 shows the result of a measurement experiment (details in §2) where we can see that, as the link speeds continue to increase through the use of higher baudrate (from 10G to 25G) and denser modulation (from 25G to 50G), optical links are becoming more susceptible to optical attenuation and thus corruption packet loss.

Optical corruption can only be remedied by physically repairing the damaged links, which can take between several hours to days [59]. During this time, the impact of corruption can only be mitigated. The current state-of-the-art approach to mitigate corruption packet loss is to disable the corrupting links while maintaining a certain minimum network capacity [54, 59]. However, this approach is not sufficient, as it is often not feasible for some corrupting links to be disabled without violating capacity constraints. Such links will continue to cause packet drops thereby negatively impacting both throughput and latency-sensitive flows. Data from Microsoft datacenters shows that up to 15% of the corrupting links cannot be disabled under realistic capacity constraints [59].

In this paper, we apply the classical loss recovery strategy of link-local retransmission for mitigating corruption packet loss in datacenter networks. Link-local retransmission has been studied extensively [9, 10, 44] and deployed widely in wireless networks [1, 2, 24, 25]. It has desirable properties such as the recovery overheads are proportional to the corruption loss rate and localized to only the corrupting link. It can achieve sub-RTT recovery and since it is agnostic to the end-hosts, it is amenable to any transport protocol including RDMA. Yet, despite these advantages, link-local retransmissions have never been deployed in the context of datacenter networks to the best of our knowledge.

We suspect that this is because deploying link-local retransmission in datacenter networks is challenging for the following reasons: first, link-local retransmission requires packet buffering while datacenter switch buffers are generally small. The problem is further exacerbated by high link speeds that will generally require more buffering. Second,
most flows in datacenter networks are short (see Figure 2),
which increases the probability of tail packet loss. Such tail
losses need to be detected and recovered at microsecond
scales to provide bounded tail FCT guarantees and meet the
stringent Service Level Agreements (SLAs) [14, 37, 52, 57].
Third, RDMA is being widely deployed in modern datacen-
ters [20, 22, 37, 58] which is more sensitive to packet reordering
than TCP [23]. Therefore, packet ordering needs to be
preserved while performing link-local retransmission.
In this paper, we show that, with modern programmable
switches, it is now feasible to implement link-local retrans-
mission in datacenter networks. Our system, Protego, can
overcome the above challenges by implementing the follow-
ing mechanisms: (1) a fast and efficient (low overhead) loss
detection and recovery protocol to keep the recovery delay
and thus the buffering requirement small (§3.1 and §3.4); (2)
a novel mechanism to detect tail packet losses quickly and
efficiently using a self-replenishing queue of “dummy packets”
without the need for a timeout (§3.2); and (3) a “reordering
buffer” at the receiver switch to maintain packet ordering
along with a PFC-based backpressure mechanism to ensure
that the buffer does not overflow (§3.3). While individually
these techniques are relatively straightforward, our key in-
sight is that their combination is sufficient to make link-local
retransmission feasible in modern datacenter networks.
Conventional wisdom says that link-local retransmis-
sions need to preserve packet ordering to prevent the transport
layer from triggering spurious loss recovery and reduction of
the sending rate [3, 4, 9, 11, 58]. We will show that in the
context of datacenters, it is not always necessary to preserve
packet ordering (§4.3). The key insight is that most flows
in datacenter networks are short [37, 46] and most flows fit
within one packet and require only 1 RTT to complete [37]
(see Figure 2). When a flow fits within a single packet, we do
not need to worry about ordering for both TCP and RDMA.
For multi-packet TCP flows, out-of-order retransmission can
still provide significant corruption loss mitigation for TCP
flows at 100G speeds even if we cannot retransmit within
TCP’s reordering window. This is because even when a TCP
flow spans multiple packets, it lasts only a few RTTs (flows
being short). This means that if there is a corruption loss,
it mostly occurs just once and thus reordering happens at
most once which has minimal impact on the FCT. To this
end, we show that a non-blocking variant of Protego (that
implement).
disabled and sent for repair as every ToR switch will lose only 1 out of 192 paths to the spine layer. However, if link B also starts corrupting packets while link A is being repaired (which can take 2 to 4 days), link B cannot be disabled since by doing so switch 1 will lose more than 25% of paths to the spine and violate the capacity constraint.

A recent study of Microsoft datacenters by Zhuo et al. showed that under realistic capacity constraints, about 15% of the corrupting links cannot be disabled [59]. Zhuo et al. hence proposed a solution called CorrOpt that finds a subset of corrupting links that can be disabled such that the impact of the remaining corrupting links is minimized. It is also possible to avoid the corrupting links via source routing or by using virtual network topologies (e.g. RAIL [60]). However, it is also not always possible to avoid faulty links without violating capacity constraints.

Why not rely on end-to-end recovery? Since there is currently no way to eliminate corruption losses, recovery is left to the end-to-end transport protocol (TCP/RDMA) by default. However, as shown in Figure 2, most flows in datacenters fit within a single packet and complete within 1 RTT under normal conditions. For such flows, under 10⁻³ corruption packet loss rate, we found that the 99.9th percentile FCTs increase by 66x and 51x when using RDMA and DCTCP, respectively (see Table 1). Furthermore, it performs metering outside the TCP's reordering window of 3 packets on Ethernet networks, SQR [45] performs link-local retransmission to handle packet corruption in (Ethernet-based) datacenter networks, LLR [41] is an NVIDIA proprietary feature that breaks an Infiniband L2 datastream into "cells" and performs cell-level retransmission for links that are not longer than 30m. In the context of datacenter networks, SQR performs link-local retransmission to recover packet loss during fail-stop link failures but SQR does not work for corrupting links. Protego hence represents a new and unexplored point in the solution design space for handling packet corruption in (Ethernet-based) datacenter networks.

Our prior workshop paper [6] investigated the potential of this general idea by implementing out-of-order retransmission within the TCP’s reordering window of 3 packets on 10G links. In this paper, we build upon that work to show that out-of-order retransmission outside the TCP’s reordering window can still be effective at 100G speeds. Furthermore, our prior work was a work-in-progress and it did not describe a complete solution that: (i) completely masks the corruption packet loss with in-order retransmission (and is hence amenable to RDMA); (ii) handles tail packet loss;
(iii) handles consecutive packet loss; (iv) works at high link speeds; and (v) can be deployed effectively on a large-scale network. Therefore, to the best of our knowledge, Protego is the first complete solution for mitigating corruption packet loss in datacenter networks using link-local retransmission.

3 PROTEGO

The corruption loss rates in real-world datacenters tend to be small (see Table 1). This makes it possible for Protego to mitigate the impact of corruption packet loss using link-local retransmission. To detect link corruption, we use a low-cost scheme that continuously monitors all optical links in the network (see Appendix C). Until it is activated, Protego lies dormant and imposes no cost on the network.

In this section, we provide an overview of Protego’s design by describing a basic link-local retransmission (LL-ReTx) scheme, the challenges of implementing LL-ReTx at line rates, and, finally, the key ideas that make LL-ReTx practical in the context of datacenter networks.

**Basic LL-ReTx.** Protego can be modeled as a protocol running between a “sender” switch and a “receiver” switch (see Figure 5). The sender adds a monotonically increasing sequence number (seqNo) to the transmitted packets and buffers a copy of the recently sent packets (in Tx buffer). These sequence numbers are used by the receiver to detect corruption packet losses. When there is no packet loss (seqNo 1-2), the receiver piggybacks the cumulative ACK information on top of reverse direction traffic (Ack2). The sender then drops the buffered copies of successfully delivered packets (seqNo 1-2). In case of a corruption packet loss (seqNo 3 in Figure 5), the receiver detects the gap in the sequence number when it receives the subsequent packet (seqNo 4). The receiver then sends a high-priority loss notification to the sender (Lost3) and the sender will retransmit the packet with seqNo 3 with high priority.

**Challenges.** While this basic LL-ReTx scheme is sufficient to achieve LL-ReTX, it is not practical in a datacenter because of the following reasons:

1. **Small buffers:** Since the switches in datacenter networks have shallow buffers, the sender needs to receive the ACKs fast enough so that it can drop the buffered packets fast enough to keep the Tx buffer usage small. If we piggyback ACKs naively, they could get delayed by an arbitrary amount depending on the reverse direction traffic.

2. **Short flows:** Since most datacenter flows are short (see Figure 2), mostly 1 packet, it is not always possible to detect the loss of such packets based on the gap in the sequence numbers. In Figure 5, if the packet with seqNo 5 belonging to a short flow is lost, then the basic LL-ReTx scheme cannot detect the same until a subsequent packet (seqNo 6) is transmitted. This can lead to high-tail FCTs.

3. **RDMA flows:** The use of RDMA in datacenters networks is now becoming increasingly commonplace [20, 22, 37, 58]. Compared to TCP, RDMA performance is very sensitive to packet ordering due to the lack of a “reordering window” [23]. The basic LL-ReTx above does not preserve the original packet ordering e.g. when seqNo 3 is lost in Figure 5.

Protego incorporates three key ideas to address these challenges to make LL-ReTx practical in datacenter networks:

1. **Self-replenishing queue of ACK packets (§3.1):** Protego implements a strictly low-priority queue with one ACK packet at the receiver switch (¶ in Figure 5). This means that there will always be packets in the reverse direction even when there is no reverse direction traffic to piggyback the ACKs.

2. **Self-replenishing queue of dummy packets (§3.2):** Protego also implements a similar strictly low-priority queue of dummy packets at the sender switch (¶ in Figure 5). The dummy packets get sent out as soon as there is no regular traffic to allow the receiver to quickly detect tail packet losses (e.g. seqNo 5 in Figure 5).

3. **Reordering Buffer without Overflow (§3.3):** To preserve packet ordering, Protego implements a reordering buffer on the receiver (¶ in Figure 5). A naive design would result in buffer overflow at today’s datacenter line rates. To prevent this, we use a PFC-based backpressure algorithm to throttle the sender when necessary.

**Scope and assumptions.** Our goal is not to completely eliminate corruption packet loss because it is too costly to achieve such a guarantee. Instead, we focus on the more modest goal of reducing the corruption packet loss rate to an operator-specified target level. To achieve the target effective loss rate, Protego also handles the case that the retransmitted copy of the packets could get lost too (§3.4). For the following sections, we assume that a corrupting link corrupts packets only in one direction which is the case with

Table 1: Corruption loss rates observed in Microsoft Datacenters [59].

<table>
<thead>
<tr>
<th>Loss Bucket</th>
<th>% Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>10⁻⁸, 10⁻⁶</td>
<td>47.23%</td>
</tr>
<tr>
<td>10⁻⁴, 10⁻³</td>
<td>18.43%</td>
</tr>
<tr>
<td>10⁻⁴, 10⁻³</td>
<td>21.66%</td>
</tr>
<tr>
<td>10⁻¹</td>
<td>12.67%</td>
</tr>
<tr>
<td>Total</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 5: Protego Design Overview.
When are no corruption losses, the sender uses the ACK packet queue is initialized with a single minimum-sized ACK packet queue that has a strictly lower priority compared to the normal packet queue (see Figure 5). Each time when the normal packet queue at the sender is empty, the "dummy" packet will be transmitted and the gap in sequence numbers can be detected immediately at the receiver.

3.1 Fast ACKs to prevent buffer overflow

When there are no corruption losses, the sender uses the ACK information from the receiver to clear its buffer by dropping the buffered packets that were successfully received. Therefore, the receiver must send the ACK information as soon as possible to keep the Tx buffer overhead low. While this can be achieved by maintaining a continuous stream of ACK packets, it would add significant overhead in the reverse direction. The overhead can be minimized by piggybacking the ACK information on regular packets, but this can cause the ACK signal to be delayed when there is no reverse traffic.

To address this problem, we introduce a novel self-replenishing ACK packet queue that has a strictly lower priority compared to the normal packet queue at the receiver (see Figure 5). The ACK packet queue is initialized with a single minimum-sized explicit ACK packet which will be sent as soon as the normal packet queue is empty. When the normal packet queue is not empty, the ACK information would be piggybacked on a normal packet. In addition, every time an explicit ACK packet is sent, we replenish the queue by adding a new explicit ACK packet back to the same queue using egress mirroring.

3.2 Detecting Tail Losses for Single-Packet Flows

Single-packet flows are common in datacenters [8, 34, 37, 47, 50]. Since losses can only be detected at the receiver from the gap in the sequence numbers, when the last packet before a short break in transmission is corrupted and lost, the receiver would not detect the loss until the packet transmission resumes. The most common approach to detect such tail losses is to employ retransmission timeouts [48]. However, in order to avoid spurious retransmissions, retransmission timeouts are required to be set conservatively considering worst-case delays [35]. To eliminate the need for a timeout, we add another self-replenishing queue at the sender with a single "dummy" packet that has strictly lower priority compared to the normal packet queue (see Figure 5). Each time when the normal packet queue at the sender is empty, the "dummy" packet will be transmitted and the gap in sequence numbers can be detected immediately at the receiver.

3.3 Reordering Buffer without Overflow

To preserve packet ordering after a corruption loss is detected, the receiver will need to buffer the subsequent out-of-order packets until the retransmission is received from the sender switch. We implement this buffering by using the re-circulation port queue as the "reordering buffer" (Rx Buffer in Figure 5). Packets received after the lost packet are buffered using recirculation, and this means that we need a way to ensure that the packets are forwarded in the right order after the lost packet is received from the sender. Furthermore, if extra copies of the retransmitted packet were to be received (§3.4), the extra copies need to be dropped (de-duplication). We achieve this by using a single state variable called ackNo which determines the correct next packet to be forwarded ahead. The (protected) packets from the sender as well as the receiver-buffered packets are continuously checked against the ackNo and sent back into the recirculation buffer until it is their turn to be forwarded. The pseudo-code for this is shown in Algorithm 1.

Since each retransmission takes a small but non-negligible delay, the reordering buffer will keep filling up with each packet loss if the subsequent packets continue to arrive at line rate, and eventually the Rx buffer would overflow. To prevent this, we employ a PFC-based pause–resume mechanism that asserts small PFC pauses on the TX MAC of the corrupting link on the sender switch. We pause only the normal packets queue (see Figure 5) so as not to affect the retransmission of the lost packets. The underlying principle is that we want to pause the transmission of the normal packet queue on the sender just enough to keep the recirculation port queue usage on the receiver switch to a small non-zero value which we set as 2 MTU (see Figure 6).

We note that there is a short delay called $t_{flight\_resume}$ before the PFC resume mechanism takes effect after the receiver decides to send a resume signal. The resumeThreshold is therefore set to a value such that during the $t_{flight\_resume}$

Algorithm 1: De-Duplication & In-Order Recovery

```
Apply to: protected, protected-reTx, recirculating rx-buffered pkts

1. if pkt.seq_no == ackNo then
   1a. forward();
   2. ackNo = ackNo + 1;
2. else if pkt.seq_no > ackNo then
   2a. mark_pkt_as_rx_buffered();
   2b. recirculate(); // will be subjected to the algo again
3. else if pkt.seq_no < ackNo then
   3a. drop(); // de-duplication
```

91.8% of corrupting links in production [59]. However, we should highlight that handling bidirectional corruption is simply a matter of running a parallel instance of Protego in the reverse direction.

Operation modes. Protego in its default mode preserves packet ordering. However, we also allow running Protego in a simple mode called ProtegoNB, where we disable the mechanism that maintains packet ordering. Our results in §4.3 show that ProtegoNB is effective in mitigating corruption packet loss for short TCP flows because of the small flow sizes as well as TCP’s support for reordering window and selective recovery.

1. does not risk a PFC storm/deadlock since Protego generates its own pause/resume frames locally on the link and not configure switch’s PFC.
Figure 6: Logical view of receiver-side ingress buffer (recirculation port queue).

time, the queue will not be fully emptied (Figure 6). Following DCQCN’s recommendation [58], we set the pauseThreshold by leaving 2MTU worth of space as hysteresis. The PFC pause/resume mechanism is described in Algorithm 2. Essentially, a pause frame is sent when the buffer level reaches the pauseThreshold, and a resume frame is sent when the buffer falls below the resumeThreshold. Since Algorithm 2 operates on a per-packet basis, we use a flag curr_pfc_state to avoid sending redundant pause/resume messages.

3.4 Mitigating Potential Retx Losses

If the link corruption rate is high, it is plausible that a retransmitted packet might also be lost. To improve the odds of a successful retransmission, the sender retransmits not one, but multiple copies of a buffered packet. Recall that our goal is not to completely eliminate corruption packet losses, but to reduce the loss rate to an operator-specified target level. Hence, the number of packets that are needed to be retransmitted to achieve this target with high probability is given by

\[ \text{reTx copies} = \left\lceil \frac{\log_{10}(\text{target loss rate})}{\log_{10}(\text{actual loss rate})} \right\rceil - 1 \] (1)

For example, if a maximum loss rate of \( \leq 10^{-8} \) is desired by the network operator and the loss rate on a corrupting link is \( 10^{-3} \), then retransmitting a single copy of the buffered packet would suffice to reduce the effective loss rate to \( 10^{-8} \). For a higher loss rate such as \( 10^{-9} \), 2 copies would be required. Since the loss rates are typically very low (Table 1), this strategy to retransmit multiple copies adds a very small overhead.

3.5 Implementation Details

Protego is implemented on an Intel Tofino programmable switch with about 1,800 lines of P4 code and runs entirely in the dataplane. For each packet to be protected, the sender switch adds a 3-byte Protego data header, consisting of a 16-bit seqNo and other metadata: the seqNo era and the packet type (original or retransmitted). To piggyback the ACK information on the reverse direction traffic, the receiver switch adds a similar 3-byte Protego ACK header. During bootstrapping, the self-replenishing queues of the dummy and the ACK packets are initialized by injecting a single minimum-sized packet from the switch control plane. All the state variables are maintained on a per-port basis using SRAM-based register memory. By default, Protego preserves ordering (§3.3) and provides a runtime option to switch to the non-blocking mode (ProtegoNB) where ordering is not preserved.

Handling seqNo Wrap-around. We handle seqNo wraparound by including an additional “era bit” along with the sequence number which toggles between 0 and 1 each time the sequence number wraps around. We perform an “era correction” when comparing two sequence numbers belonging to different eras, where we subtract a constant \( N/2 \) from both the sequence numbers (\( N \) is the sequence number range). This works correctly as long as the two different-era sequence numbers are not more than \( N/2 \) apart.

Handling consecutive packet losses. To decide which packets to retransmit, the sender switch maintains a lookup table reTxReqs which is updated by the receiver (Figure 5). When consecutive packets are lost, multiple entries in reTxReqs need to be updated simultaneously by the loss notification packet. If reTxReqs is implemented as a single register, such a simultaneous update is not possible due to hardware limitations. Therefore, we implement reTxReqs across multiple 1-bit registers (details omitted for brevity) where the number of registers required is equal to the maximum number of consecutive packets lost. In our current implementation, we provision 5 1-bit registers (across 2 pipeline stages) which based on our measurement results (details in Appendix B.2) can handle 99,9999% of corruption loss events at an unreasonably high packet loss rate of 5%.

Preventing transmission stalls. In spite of our best efforts, there is still a small but non-zero probability that a retransmission will not be successful. Because we buffer packets at the receiver until all corrupted packets are received, this could stall the transmission indefinitely and cause the Rx buffer to overflow. To handle this rare but potentially fatal event, we implement a timeout called ackNoTimeout at the receiver. If a retransmission does not occur within the timeout period, the receiver ignores the lost packet, increments the ackNo and continues with the remaining packet transmissions. The ackNoTimeout is set to a value greater than the maximum expected delay in receiving a retransmission after a packet has been found to be lost (details in Appendix B.1). To update the ackNo at the receiver when there is an ACK timeout (see §3.3), we use periodic packets.
from the switch’s packet generator for timekeeping [31]. In our implementation, we set the rate of these timer packets to 10 Mpps (∼1% of switch’s pipeline processing capacity).

**Packet Generation.** To create multiple copies of a buffered packet during retransmission (in case of a high loss rate), the sender switch uses the multicast primitive. Upon detecting a loss, the receiver switch uses ingress mirroring to generate the loss notifications. Whenever PFC pause/resume packets need to be sent by the receiver, we modify the timer packets and send them to the sender switch.

### 3.6 Repairing Corrupting Links in Practice

Recall that Protego is activated on a link only when the link is found to be corrupting packets (§3). However, if we only enable Protego and do nothing to repair the corrupting links, then over a long period of time (∼1-2 years), we might end up having Protego activated on the majority of links in a large datacenter network. Therefore, as a long-term strategy for maintaining the network, periodically, we will need to bring down the corrupting links so that they can be repaired.

A simple way to do this is to run an algorithm like CorrOpt [59] to safely schedule Protego-enabled links for repair without violating capacity constraints. In particular, when a link starts corrupting packets, we immediately enable Protego on it to reduce the effective loss rate to an acceptable rate. Then we run CorrOpt’s fast checker algorithm to check if the link can be safely disabled and scheduled for repair. If so, we disable the link and schedule for repair. Otherwise, the link continues to operate with corruption while Protego mitigates the impact on application performance. As links get enabled again after their repair is complete, we run CorrOpt’s optimizer algorithm to see if any of the Protego-enabled corrupting links can be safely disabled and scheduled for repair. What this joint strategy also demonstrates is that instead of being in competition with previously proposed algorithms, Protego is complementary to them.

### 4 EVALUATION

In this section, we present our evaluation results for Protego and ProtegoNB (out-of-order recovery). In particular, we seek to answer the following questions:

1. How effective is Protego at masking the corruption packet losses? Are we able to reduce the effective loss rate to the operator-specified target as desired? And what is the corresponding reduction in link speed?
2. How well does Protego handle tail packet loss and improve FCTs for short and single-packet flows?
3. How does Protego’s performance compare with Wharf [21], the state-of-art link-local FEC solution?
4. How much buffering does Protego need and what are the associated overheads and costs of deploying Protego?

(5) When deployed in a large-scale network, how effective is Protego in reducing the corruption packet loss and improving the overall network capacity?

**Testbed Setup.** We use the testbed setup shown in Figure 7, where sw2 and sw6 are connected by an OM4 grade fiber optical fiber link. Depending on the experiment, links are either all 25G or all 100G. sw2 and sw6 act as the Protego sender and receiver respectively and we restrict their recirculation buffers to 200 KB. Following the methodology used in [60], we introduce corruption packet loss on the link between sw2 and sw6 using a VOA. We set Protego’s target loss rate2 to $10^{-8}$ and the number of retransmitted packet copies is then determined by Equation 1 depending on the actual loss rate.

Using the switch control plane, we poll the port counters for ports denoted by A, B, C and D in Figure 7. These counters enable us to measure the sending rate/throughput of an endpoint sender, the actual loss rate incurred due to the VOA, and the effective loss rate and link speed achieved by Protego. We also poll the queue occupancies on sw2 and sw6 using the local control plane.

The servers are equipped with Intel Xeon Silver/Gold CPUs, 128 GB memory, NVIDIA CX6-DX NICs (25G/100G) and run Linux kernel 5.4.0-91-lowlatency on Ubuntu 20.04.3. For our experiments, we use kernel-based DCTCP and NIC-based RoCEv2 (RDMA) transports. For TCP, TSO, SACK, RACK-TLP and ECN (100 KB marking threshold [15]) are enabled and $RTO_{min}$ is set to 1 ms. The network RTT for a TCP sender is ∼30 μs. For RoCEv2, we use a one-sided RDMA_WRITE operation using NIC-based reliable delivery (RC [42]) which we found to have a $RTO$ of ∼1 ms.

**Parameters.** Protego uses 3 parameters: `ackNoTimeout`, `resumeThreshold`, and `pauseThreshold`. As discussed in §3.5, we set the `ackNoTimeout` to 7.5 μs and 7 μs as we found the maximum retransmission delays to be 6 μs and 5.5 μs for 25G and 100G links respectively. For the `resumeThreshold` (§3.3 and Figure 6), we measured the maximum $t_{flight}$ values to be 1.9 μs and 1.6 μs for 25G and 100G links respectively. Therefore, we conservatively set the `resumeThreshold` at 40 KB and 37 KB for 25G and 100G links respectively as the recirculation-based buffer drains at 100G. Since we use a fixed hysteresis of 2 MTU, the `pauseThreshold` is $10^{-8}$.

---

2For MTU-sized packets, a loss rate of $10^{-8}$ corresponds to a bit error rate (BER) of $10^{-12}$ which is considered a healthy/non-corrupting link [60].
Our high-level goal is to mask the corruption packet losses from the transport layer. While we showed in §4.1 that Protego can reduce the effective loss rates, what matters is the net impact on transport protocols. To understand the impact of Protego, we send single flow TCP traffic from h4 to h8 using iperf with all links set to 25G. We evaluate three different TCP variants: CUBIC, DCTCP, and BBR, as they use congestion loss, ECN, and delay as congestion signals respectively. We consider BBR to be representative of delay-based transport protocols, since the implementations for TIMELY [38] and Swift [32] are not readily available.

In each experiment, we start the setup with no corruption loss. At the 2 second mark, we introduce a loss rate of $10^{-3}$ on the link, and approximately 5 seconds later, we enable Protego. We plot the results for DCTCP in Figure 9a. The effective link speed in the figure is measured separately by sending a line rate UDP flow under the same experiment conditions. We see that the throughput is reduced sharply once corruption losses are introduced. Upon enabling Protego, the corruption losses are eliminated and the throughput returns to a level comparable to the effective link speed. We also notice that the slightly lower effective link speed leads to a build-up in the flow’s buffer at the sender switch (shown as “qdepth”) triggering ECN marking. This result also demonstrates that since Protego only deals with packets transmitted on the link, it works well even if the link has congestion. Overall, we see that Protego’s backpressure mechanism is effective at keeping its receiver-side buffer occupancy (labelled as ”rx buffer”) low. We observe similar results with CUBIC and BBR (see Appendix B.3).

**Backpressure Not Considered Optional.** In Figure 9b, we also plot the results when the PFC-based backpressure
mechanism disabled. We now see a large number of end-to-end retransmissions because the reordering buffer (Rx buffer) periodically builds up and overflows. In fact, the observed packet losses after enabling Protego are so severe that the random corruption packet losses in the period between 2 and 8 seconds are barely visible in Figure 9b. The throughput is also lower compared to the earlier results shown in Figure 9a. In other words, the PFC-based backpressure mechanism is critical for ensuring that the buffering at the receiver switch works as intended.

4.3 Tail Packet Loss and Short Flows

One-packet Flows. To evaluate how effectively Protego handles tail packet losses, we measure the FCT of 143 B DCTCP and RDMA_WR flows in our testbed with all links set to 100G while introducing a corruption loss rate of $10^{-3}$. 143 B is the most frequent flow size in the Google all RPC workload [50]. It is clear from our results in Figure 11 that both Protego and ProtegoNB are able to mask the corruption losses so effectively that the performance at $10^{-3}$ loss rate becomes indistinguishable from the case when the link is lossless. Protego and ProtegoNB achieve the same performance since we do not need to worry about ordering in case of single packet flows. We note that the result in Figure 11 is also representative of all other flow sizes for workloads in Figure 2 that fit within a single packet.

Longer (multi-packet) Flows. Next, we repeat the experiment with 24,387 B-sized flows which is the most frequent flow size in the DCTCP web search workload [3]. We plot the results when using DCTCP, BBR and RDMA_WRITE transports in Figure 10. We can see that the lines for Protego and no loss mostly overlap. While BBR is mostly agnostic to packet loss, this experiment shows that corruption packet loss does affect the FCTs of short BBR flows and therefore mitigating corruption loss is necessary for BBR and similar rate-based/loss-agnostic transport protocols. In Figure 10, we also see that for RDMA, ProtegoNB provides no improvement over the loss case other than preventing RTO by handling tail packet losses. This is because RDMA’s NIC-based reliable delivery has no reordering tolerance and ProtegoNB does not cause any reordering when it recovers the tail packet loss. On the other hand, for DCTCP and BBR, ProtegoNB performs nearly as well as Protego except at very high percentiles (> $99.9^\text{th}$) where it performs marginally worse.

**Why does ProtegoNB perform so well?** For single-packet flows, it is unsurprising that the transport layer performance is the same for both Protego and ProtegoNB. For longer flows, we found that since TSO is enabled, packet bursts travel at near line rate (100G) and ProtegoNB is not able to perform out-of-order recovery within TCP’s reordering window of 3 packets. However, since the flows are short, this does not significantly affect the FCT for two reasons: (i) corruption often happens among the last 3 packets for short flows where there is no reduction in cwnd as the TCP sender does not receive sufficient SACKed bytes ($\geq 3$ MSS) while ProtegoNB performs a sub-RTT but out-of-order recovery; and (ii) in cases when there is cwnd reduction, since the flows are short, it does not significantly affect the FCT. For BBR, there is no reduction in sending rate since BBR is loss-agnostic. However, BBR still benefits from ProtegoNB by avoiding 1 RTT delay as well as TCP end-host stack latencies involved in end-to-end recovery.

In summary, both Protego and ProtegoNB improve the 99.9th percentile FCT for single packet DCTCP and RDMA flows by 51x and 66x respectively. For longer flows, the 99.9th percentile gains for Protego are 19x for DCTCP and BBR, and 39x for RDMA. While ProtegoNB performs similar to Protego for longer TCP flows (up to 99th percentile), it provides little benefit in case of reordering-sensitive RDMA but does eliminate the long tail FCTs due to RTOs.

---

**Figure 10:** Top 5% FCTs for 24,387B flows (17 pkts) on a 100G link.

**Figure 11:** Top 1% FCTs for 143B flows on a 100G link.
We used control plane APIs to measure the packet buffer usage which we plot in Figure 12 for 25G and 100G links running at three different loss rates. The key takeaway from these results is that at 25G, the TX and RX buffer usage for Protego are at most 3.6 KB (~2 MTU) and 60 KB respectively for all evaluated loss rates; at 100G, the TX and RX buffer usage are both at most 90 KB. ProtegoNB requires no RX buffer, while its TX buffer requirement is same as Protego at 25G and about 3x lower (24.4 KB) at 100G. This is because ProtegoNB has no PFC-based backpressure mechanism that could potentially delay the ACKs. To put these numbers in context, 100G datacenter switches have 16-42 MB of packet buffer [53]. In other words, the required buffering to deploy Protego is negligible for modern switches.

Protocol Overhead. Protego adds a 3-byte header to each packet in both forward and reverse (ACK) directions. Since standard MTU-sized frame is 1,538 octets on wire, this overhead amounts to a ~0.2% degradation of link capacity and occurs only when Protego is activated. Both the dummy packets and explicit ACK packets do not add any overheads as they are transmitted only when there is no regular traffic.

Recirculation Overhead. Across 3 loss rates and 2 link speeds, we found the worst case recirculation overhead to be 0.664% of the switch pipeline’s processing capacity (more details in Appendix B.4). ProtegoNB has the same recirculation overhead on the sender switch but zero on the receiver switch. The key takeaway is that recirculation takes up less than 1% of the switch pipeline’s processing capacity, and thus the overhead is negligible for modern switches.

Dataplane Resources. Protego needs to maintain state in the dataplane on a per-port basis and uses stateful ALUs (SALUs) for stateful operations. With state provisioned for 256 ports, Protego requires only ~9% of the total SRAM memory and uses ~25% of the available SALUs. While 25% might seem high, we note that SALUs are not used by other switch forwarding functions which are typically stateless.

We note that, except for the dataplane resources, the above overheads are per Protego-enabled link. However, the results
Table 3: TCP CUBIC goodput (Gb/s) on a 10G Link

<table>
<thead>
<tr>
<th>Loss Rate</th>
<th>0</th>
<th>10^{-3}</th>
<th>10^{-4}</th>
<th>10^{-5}</th>
<th>10^{-6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>9.49</td>
<td>9.48</td>
<td>8.01</td>
<td>3.48</td>
<td>1.46</td>
</tr>
<tr>
<td>Wharf</td>
<td>n/a</td>
<td>9.13</td>
<td>9.13</td>
<td>9.13</td>
<td>7.91</td>
</tr>
</tbody>
</table>

from our large-scale simulation (§4.7) suggest there could be no more than 2-4 Protego-enabled links on a switch pipe.

4.6 Comparison with Wharf

Link-local FEC is a natural alternative to link-local retransmissions and therefore we compare Protego with Wharf [21], the state-of-the-art link-local FEC scheme. We were not able to reproduce Wharf’s results experimentally because we did not have access to the required FPGA hardware. In Table 3, we reproduce Wharf’s results numerically by picking the Wharf FEC parameters that gave their best-reported goodput for each loss rate (c.f. Figure 8 in [21]). In our experiments, we used the same experimental setup as Giesen et al.: 1G link, TCP CUBIC, and Tofino-based random packet dropping. Our results show that both Protego and ProtegoNB compare favorably at all loss rates. For ProtegoNB, we observed that it retransmits within TCP’s reordering window for majority of times and thereby prevents the TCP sender from reducing its cwnd below the network BDP.

4.7 Effectiveness in large-scale deployment

In this section, we use simulation to evaluate the effectiveness of Protego when deployed in a large datacenter network network. We use the same methodology that was used to evaluate CorrOpt [59] and compare vanilla CorrOpt with Protego + CorrOpt.

**Setup.** We contacted the authors of CorrOpt [59] for details on their evaluation setup. However, due to confidentiality reasons, they were unable to provide us any traces, source code or topology information. Therefore, we re-implemented their evaluation methodology in about 3K lines of Python code. For the topology, we use the state-of-the-art Facebook fabric [5] (see Figure 4) datacenter network with about 100K switch-to-switch optical links (all 100G) and 1:1 oversubscription ratio. For link corruption trace, we implemented a trace generator that uses the corruption loss rate and link spatial distribution data from Microsoft’s datacenters [59], and a per-link Weibull distribution with a mean-time-to-failure (MTTF) of 10K hours [36] (details in Appendix D). When activated, Protego performs ordered retransmission and the link’s effective speed is as per Figure 8. We assume that when sent for repair, 80% of the corrupting links are repaired in ~2 days while the remaining take ~4 days [59].

![Figure 13: 1 week snapshot of simulation results for FB fabric topology (100K optical links).](image)

![Figure 14: CDF of (a) Gain in the total penalty; and (b) Decrease in least capacity per pod; for Protego + CorrOpt compared to vanilla CorrOpt (1 year simulation).](image)
penalty (gain = 1) as all corrupting links are disabled successfully. However, for the remaining 65% of the time and for nearly all times with 75% capacity constraint, the combined solution offers significant benefits while causing very little reduction in the per pod’s capacity to the core (Figure 14b).

Overall, our results demonstrate that when augmented with Protego, CorrOpt can reduce the total penalty by orders of magnitude while allowing the network to be operated at higher capacity constraints, with a minimal reduction in network capacity.

5 DISCUSSION & FUTURE WORK

In this section, we discuss a few corner cases, address the current implementation constraints with next generation programmable hardware and discuss future extensions.

Handling multiple corrupting links on the same switch. In our simulations (§4.7), we observed that in the worst case there could be 2 and 4 concurrently Protego-enabled links per switch pipeline for capacity constraints of 50% and 75% respectively. Switch pipelines support ~2 internal recirculation ports per pipe [40] and more recirculation ports can be added by running any free ports in loopback mode. Nevertheless, the basic question that remains to be answered is the following: how can we use ~2 recirculation ports to buffer packets for >2 Protego ports? We believe that this could be plausible as datacenter link utilization is bursty and not all ports run at 100% capacity at the same time [56]. Also, since Tofino2 can likely implement retransmission without recirculation, Tofino2 can naturally support multiple corrupting links.

Implementing Protego with Tofino2. In our measurements (detailed in Appendix B.1), we found that Protego takes up to 5.25 µs to recover an MTU-sized (1,538 B on wire) packet on a 100G link. Given that it takes only about ~123 ns to serialize 1,538 bytes on a 100G link, this delay is surprisingly long. It turns out that this large delay is an artifact of our current recirculation-based buffering on the Intel Tofino. Tofino2 [33] offers new advanced flow control primitives that could be used to pause/unpause as well as achieve credit-based scheduling of a queue entirely in the dataplane. These primitives could in theory allow us to implement retransmission without recirculation, but this thesis remains to be validated.

Protego vs. ProtegoNB. Our results in §4 suggest that ProtegoNB is generally more scalable to higher loss rates and link speeds and incurs less overheads compared to Protego. While ProtegoNB performs comparably to Protego for TCP (upto 99th percentile), the difference is more significant for RDMA NIC-based reliable transport. Depending on the application mix and the desired SLA guarantees, a network operator could do a runtime configuration to run either Protego or ProtegoNB on a corrupting link. In fact, while currently not implemented in our prototype, it is reasonably straightforward to allow both Protego and ProtegoNB to run simultaneously on a corrupting link, each protecting a different class of traffic with different ordering guarantees.

Incremental Deployment. Protego is suitable for incremental deployment as switches are upgraded over time in a network. As discussed in §2, not all links are equal, and therefore network operators can prioritize deploying Protego for links which if disabled can significantly reduce the per-ToR paths. The exact partial deployment strategy that yields maximum benefits remains as future work.

Higher Link Speeds. Protego is agnostic to the overall scale of the network as it works locally on the link between adjacent switches. The question is whether Protego would continue to work as link speeds continue to grow. In principle, ProtegoNB would work well for higher link speeds of 400G and above due to its lower overheads and better scalability. Protego, on the other hand, might achieve a proportionally lower effective link speed and higher buffer overhead if the switch pipeline latency hugely dominates the retransmission delay. However, we believe that with a Tofino2-based implementation and further dataplane optimizations, Protego should still achieve good performance with low overheads. We plan to investigate this once the hardware becomes available to us.

Reordering tolerance in modern transport protocols. Recently, RFC8985 [13] has introduced a new feature called the "reordering window adaptation" in the Linux TCP stack. Also, RoCEv2’s NIC-based reliable transport has a new “selective repeat” feature [43] that allows more efficient selective retransmission than Go-back-N recovery. We plan to investigate the implication of these new features for ProtegoNB.

6 CONCLUSION

To the best of our knowledge, we are the first validate that a combination of simple techniques can make link-local retransmission practical in modern datacenter networks. With Protego, network operators can work with corrupting links with moderate loss rates (between 10^-3 and 10^-5) at a marginally reduced link speed and with little overhead. Since Protego is amenable to incremental deployment, deploying Protego with CorrOpt will allow network operators to not only reduce the network-wide corruption loss rate, but also operate networks at higher capacity constraints that were not previously feasible. Overall, we believe that we have made a strong case that link-local retransmission is both practical and effective for modern datacenter networks.

Ethics statement: This work does not raise any ethical issues.
REFERENCES


---

A PROTOCOL DETAILS

In this Appendix, we provide some details that might be helpful for understanding our complete implementation of Protego, but which are not essential for understanding the key ideas and contributions of our work.

A.1 Loss Detection & Notification

In Figure 15, we list the state variables maintained by the sender and receiver switches and the different packets that are exchanged. The sender maintains a monotonically increasing seqNo while the receiver records the latest received seqNo as latestRxSeqNo. A copy of the latestRxSeqNo is also maintained at the sender, which the receiver keeps updating. The sender also maintains a lookup table called reTxReqs, which records the sequence numbers of the packets for which retransmission is requested.

For each packet that is transmitted on the corrupting link (protected packet), the sender adds the seqNo to the packet (using a custom header) and increments it by 1. The sender uses egress mirroring to also make a copy of the packet along with the added sequence number and buffers it until the receiver notifies that the packet was received successfully. On the receiver, when a protected packet is received, it updates the latestRxSeqNo to the seqNo in the packet and also sets the pendingAck to 1. pendingAck set to 1 denotes that the copy of latestRxSeqNo on the sender is yet to be updated.

No Loss Scenario. When there are no corruption packet losses, the latestRxSeqNo on the receiver would increase by 1, each time a protected packet is received. On every update of the latestRxSeqNo, the receiver must update the latestRxSeqNo on the sender as soon as possible so that the sender can drop the buffered packets that are successfully delivered. This timely update of the latestRxSeqNo on the sender is critical to ensure that Protego’s use of the packet buffer at the sender is kept to a minimum.
Loss Scenario. When a protected packet(s) gets corrupted and dropped by the receiving MAC, the receiver observes that the latest RxSeqNo is incremented by more than 1. On noticing this, the receiver activates a LossDetection() routine. In this routine, the receiver generates a new packet called “Loss Notification” which contains information about the missing sequence number as well as the latest RxSeqNo. This loss notification packet is sent to the sender through a high-priority queue (see Figure 5) to ensure timely recovery. On reaching the sender, the lookup table reTxReqs (Figure 15) is updated with the sequence numbers of the packets that need to be retransmitted.

A.2 Sender-side Buffering & Retransmission

For each packet that is sent on the corrupting link, the sender switch adds a monotonically increasing seqNo and uses egress mirroring to create a copy of the packet for buffering. The packet buffering on the sender switch is realized through recirculation. Specifically, the buffered copy of the protected packet is sent to the recirculation port of the switch dataplane pipeline. At the same time, as described in §A.1, the receiver switch keeps the latest RxSeqNo on the sender switch updated and additionally updates the lookup table reTxReqs in case of a corruption packet loss. Each time the buffered packet completes a recirculation loop, the sender switch applies the logic shown in Figure 16 to the packet’s sequence number. Essentially, if the buffered packet’s sequence number is less than or equal to the latest RxSeqNo, the sender switch checks the reTxReqs lookup table to see if a retransmission is requested for that sequence number. If so, the packet is retransmitted through a high-priority queue (see Figure 5) or the packet is dropped otherwise. If a packet is retransmitted, its sequence number is cleared in the reTxReqs table. If the buffered packet’s sequence number is greater than the latest RxSeqNo, then we do not know yet if the packet was successfully received or not and therefore the sender switch continues to buffer the packet through recirculation.

Figure 16: Sender-side buffering and Retransmission.

![Diagram showing the logic of sender-side buffering and retransmission.]

Figure 17: Delay observed by Protego receiver switch to receive retransmission from the time the loss was detected.

B ADDITIONAL EXPERIMENTS AND RESULTS

B.1 Parameter Tuning

In this section, we describe how we derive the appropriate values for three parameters used by Protego: ackNo Timeout, resumeThreshold, and pauseThreshold.

Recall that when Protego preserves packet ordering (default mode), the ackNoTimeout prevents Protego from stalling in the event that a lost packet is never recovered (§3.3). Therefore, ackNoTimeout needs to be set to a value larger than the expected maximum retransmission delay. To estimate the retransmission delay, we measured the time from when the receiver switch detects packet loss to when it successfully receives the retransmission from the sender switch. Since high-priority queues are used for loss notification and retransmission, this retransmission delay is a function of the switch pipeline latencies, the link speed, and the number of retransmitted copies. If more than one copy is retransmitted (for higher loss rates), then the worst case retransmission delay is when only the last copy is received. In Figure 17, we plot the distribution of the retransmission delays for ~1 million loss recoveries for 1,518 B packets. We conservatively set the ackNoTimeout to 7.5 µs and 7 µs for 25G and 100G, respectively. A larger ackNoTimeout leads to a slightly longer stall in transmission, but only in the unlikely event that the original packet and all retransmitted copies are lost.

The PFC-based backpressuring mechanism on the Protego receiver uses the pauseThreshold and the resumeThreshold (see §3.4 and Figure 6). Recall that a PFC resume frame is sent when the reordering buffer drops below the resumeThreshold. If resumeThreshold is set too small, the receiver recirculation buffer will be empty before the sender switch successfully resumes transmissions. Hence, we set resumeThreshold to a value that is larger than the amount of data that would drain from the buffer during the time from when the receiver sends a PFC resume frame to when the receiver starts receiving traffic again. We refer to this time as t_flight_resume. t_flight_resume is independent of the corruption loss rate and depends only on the link speed and switch pipeline latencies. We measured the maximum t_flight_resume values to be 1.9 µs and 1.6 µs for 25G and 100G links respectively. Since the recirculation-based buffer drains at 100G, we set
B.2 Consecutive Corruption Packet Loss

In Figure 18, we plot the distribution of the number of consecutive packets lost that we measured by setting the VOA to induce unreasonably high loss rates of 1% and 5%. Based on Figure 18, our current implementation provisions for handling 5 consecutive packets lost using 5 1-bit registers.

B.3 Impact on CUBIC and BBR transports

In this section, we present the results of the same experiment as in §4.2 but with CUBIC and BBR transports. In Figures 19a and 19b, we plot the results for CUBIC, and BBR respectively. The effective link speed in these figures is measured separately by sending a line rate UDP flow under the same experiment conditions.

CUBIC. In Figure 19a, we see that at 10\(^{-3}\) corruption loss, the throughput for CUBIC reduces sharply once corruption losses are introduced. Upon enabling Protego, the corruption losses are nearly eliminated and the throughput returns to a level comparable to that before packet corruption was introduced. We also notice that there is a build-up in the flow’s buffer at the sender switch (shown as “qdepth”) due to the reduced effective link capacity. CUBIC being loss-based, we can also see congestion loss happening once Protego is enabled. Note that Protego only protects and retransmits the packets that are sent out on the corrupting link and is not affected by any congestion loss happening due to the overflowing of the normal packet queue.

BBR. Since BBR is mostly agnostic to packet loss, we see in Figure 19b that it suffers minimal degradation when corruption loss is introduced\(^4\). Nevertheless, it seems that once Protego is enabled, we still see a small increase in the observed throughput.

These results show other than ECN-based DCTCP, even loss-based and delay-based congestion control protocols work correctly with Protego.

\(^4\)We only ran BBR on a 10G link instead of a 25G link because BBR became CPU-limited when we tried to run the experiment on a 25G link, and it was not able to fully saturate the link.

B.4 Overheads

Recirculation Overhead. In Table 4, we show the recirculation overhead for Protego at both the sender and the receiver switches in terms of the percentage of the switch pipeline’s processing capacity. ProtegoNB has the same recirculation overhead on the sender switch but zero on the receiver switch. The key takeaway is that recirculation takes up less than 1% of the switch pipeline’s processing capacity, and thus the overhead is negligible for modern switches.

C MONITORING LINKS FOR CORRUPTION

To detect corrupting links, we implemented corruptd, a daemon which runs at the local control plane of the programmable switches.

Detecting Corrupting Links. corruptd periodically polls the driver (in this paper, we configure the interval as 1
second) to extract the switch port RX statistics, specifically, framesrxok and framesrxall. We maintain a moving window of 100M frames to compute the link loss rates, given by
\[ L = \frac{\text{framesrxok}}{\text{framesrxall}}. \]
When \( L \geq 10^{-8} \) for any particular link, the upstream transmitting switch will be notified to activate Protego.

**Notification and Activation.** For scalability, corruptd daemons communicate through a publish-subscribe (PubSub) pattern using Redis. Each daemon subscribes to link corruption notifications relating to the local switch’s links. Upon receipt of a notification, corruptd pushes corresponding data plane match-action table entries to activate Protego for the corrupting link depending on the target and the actual loss rates (see Equation 1).

### D LINK CORRUPTION TRACE GENERATION

A link corruption trace is essentially a time series of link corruption events where a link corruption event denotes which link started to corrupt packets and at what loss rate. To determine the time at which a link would start corrupting packets, we assume a per-link 1-parameter Weibull distribution with a constant shape parameter (\( \beta \)). This is because the location parameter of the Weibull distribution (\( \gamma \)) is zero since it is not guaranteed that all links in a large warehouse-scale datacenter would not start corrupting packets during a certain initial period. Also, the shape parameter (\( \beta \)) is equal to 1, since the corruption is purely caused by random external events such a connector contamination, fiber bending, etc. Therefore, the per-link Weibull PDF that determines the time until a link’s next failure is given by

\[ f(t) = \frac{1}{\eta} \times e^{-\left(\frac{t}{\eta}\right)} \]  

(2)

where the parameter \( \eta \) is the mean-time-to-failure (MTTF) of a link. A study by Meza et al. [36] showed that for fiber links from different vendors considered in their study, the mean time between the link faults was at most 10,000 hours. We conservatively use the value of 10,000 hours as the MTTF (\( \eta \) in Equation 2) since Meza et al. did not specifically consider only intra-datacenter links. What this means is that, on average, it would take 10,000 hours (or 1.15 years) for a fiber link to start corrupting packets from the time it was last repaired.

To generate the trace, we first draw samples from the Weibull distribution independently for each link to determine the times at which each link would start corrupting packets. This gives us the various times of the corruption events and the link involved in each corruption event. Then for each corruption event, we use the corruption loss rate distribution from CorrOpt (c.f. Table 1 in [59]) to determine the loss rate. This list of corruption events sorted by time forms the link corruption trace. We note that the trace generated using the above methodology has a nearly random spatial distribution of simultaneously corrupting links which matches the observation by Zhuo et al. [59] in production datacenters.