# Window Based Monitoring: Packet Drop Detection in the Network Data Plane

**(extended abstract of the MSc dissertation)**

Afonso de Paiva e Pona Corte Real Gonçalves
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professor Fernando Ramos and Professor Luís Rodrigues

## Abstract

Detecting network anomalies, such as packet loss, is becoming an increasingly important task to network operators, as applications are becoming more and more performance sensitive. Several solutions aim to detect these events as soon as they occur, as well as to disclose where they are taking place. Unfortunately, some of them incur in unacceptable overhead while others have to sacrifice coverage in order to cope with the increasing traffic intensity. Software Defined Networks and the Programmable Data Plane are relatively recent technologies that allow network operators to configure how switches process packets, which opens the door to efficient monitoring solutions. In this thesis, we develop WBMon, a passive solution that leverages the Data Plane programmability to perform an inter-switch coordination algorithm that detects packet drops in arbitrary paths at line speed. Additionally, we employ a Failure Inference Algorithm (NetBouncer [9]) to enable localizing the links responsible for packet drops. Our evaluation shows that WBMon is able to detect every packet drop in less than 2ms, which allows to detect short-lived failures.

**Keywords:** Drop Detection; SDN; Programmable Switches.

## 1 Introduction

The task of monitoring the operation of computer networks is a key component to ensure the performance of current distributed systems, as it allows to gather information that can be used for planning the network evolution, and to detect anomalies, such as faults and intrusions.

In our work we are interested in the use of network monitoring for anomaly detection and, in particular, to detect links that experience excessive packet loss rates. We survey the main techniques that can be used to detect network anomalies, giving emphasis to techniques that leverage the availability of programmable switches to increase the accuracy and efficiency of this task.

From previous work we have identified two main strategies to detect faulty links. The first strategy involves the active exchange of probe traffic among different "observation-points" placed in strategic locations in the network; the data collected by these observation points can be correlated to give hints on the location of eventual faulty links and/or switches. The second strategy uses programmable network switches to detect faulty links in a passive manner, without the need to inject probe traffic; unfortunately, it only works for links that connect directly two programmable switches, and cannot be trivially applied to networks that have a combination of programmable and non-programmable switches.

Based on these observations, we propose a new strategy that combines and extends the two techniques above. First, we aim at using programmable switches to detect packet loss in the path connecting these switches, even if the path includes multiple links and non-programmable switches, such that a few programmable switches can be used as *passive* observation points. Then, we plan to correlate the information collected by these switches to *narrow* the set of potential faulty physical links. We experimentally show that our solution is able to detect every packet drop in computer networks and that, in certain topologies, it is able to locate 99% of the faulty links while using only 60% of monitoring switches.

## 2 Background

Network monitoring is the task of continuously extracting information regarding the operation of a computer network, in order to better understand how it is being used and to detect potential anomalies, faults, attacks, or other impairments to its correct operation. The scale of current networks, combined with the heterogeneity of equipment and protocols that can be used, make the task of performing network monitoring extremely challenging. Fortunately, some technological advances in the networking architecture and hardware, including Software Defined Networks and programmable switches, can now be leveraged to make network monitoring more accurate and efficient.

### 2.1 Software Defined Networks

One of the main tasks of network routers and switches is packet forwarding, which consists in receiving a packet from an ingress link and forwarding it to the next hop towards the destination via an egress link. To perform this task, the switch needs to maintain a *forwarding table*, that specifies which egress link should be selected when forwarding a packet. The other main task is to execute the logic required to populate the forwarding table, typically a distributed routing protocol,

such as RIP, OSPF, or BGP. The former task is designed to execute in the *Data Plane*, and the latter in the *Control Plane*.

One of the most important advantages of running the Control Plane in every router is the autonomy and decentralization it provides: routers that execute the Control Plane can coordinate with each other to populate their forwarding table without being dependent of other additional components. However, distributed routing protocols are notoriously complex and difficult to debug. Additionally, routing equipment was typically provided by vendors with proprietary implementation of a fixed set of routing protocols that could not be easily adapted or expanded.

#### 2.1.1 SDN.

Software Defined Networking is an architectural model that decouples the Data Plane from the Control Plane. In this model, switches only implement packet forwarding and export an interface that allows an external component to populate the forwarding table. The Control Plane is executed in this logically centralized entity, the *controller*, that decides how to populate the forwarding table of every switch.

Having a single point of control has several relevant advantages. Namely, it makes the control logic simpler to program and easier to verify, and facilitates network configuration. Moreover, it allows to have a global view of the network, and consequently to compute optimized solutions for the entire network, which was not possible to do with a distributed Control Plane.

OpenFlow [5] is a standard that specifies the interface between the controller and the switches and that allows the controller to remotely update the forwarding tables. According to this standard, each switch maintains a Flow Table that keeps a list of Match-Action rules, each consisting of a matching and an action part. The matching rule corresponds to a set of conditions that must be met to activate the action part to that packet. The action, in turn, defines what should be done to the matched packet. Typical actions are dropping a packet or forwarding it to one or multiple egress ports, but can also include changing or pushing header fields.

#### 2.1.2 Programmable Data Plane.

Processing packets according to OpenFlow rules requires switches to be able to extract the information required to match those rules. For this reason, before performing the Match-Action phase, switches execute a Parsing stage to extract that information.

The emergence of Reconfigurable Match-Action Tables (RMT) allowed switches to parse arbitrary headers and define match-action rules programmatically. This further led to the development of architectures and languages that were able to leverage this capability. The P4 programming language can now be used to specify exactly how to parse packet headers and the Match-Action rules to be applied in the forwarding pipeline. Moreover, the P4 language creates an abstraction that completely separates the Control Plane from the Data Plane, as it can be compiled into numerous targets, such as ASIC switches, Field-Programmable Gate Arrays (FPGA), etc.. The language includes the P4 Runtime API, a gRPC-based mechanism that allows a remote controller to update the tables of any P4-programmable target.

Switch programmability allowed to redesign multiple network solutions, as it granted network insights that proved to be extremely useful in tasks such as network debugging and monitoring.

### 2.2 Packet Reordering

Whenever two machines are communicating over a network, it is not guaranteed that the packets arrive in the same order as they were sent. In this section, we formalize the reordering concepts used along this document.

Let's start by assuming that the sending machine numbers packets in the order they are sent, and that the receiver records the highest number it has received. For simplicity, we call *sequence number*, or *sn* to the number representing the packet ordering, and *max_sn* to the highest *sn* stored by the receiver. Note that these sequence numbers are different from the ones used in TCP. While TCP sequence numbers refer to the *byte* ordering, the sequence numbers used in this work are relative to the *packet* ordering, regardless of the size of each packet.

In this work we define reordered packets in accordance to RFC 4737. A packet is reordered if, at the time of its arrival, its sequence number is smaller than *max_sn*. In other words, reordered packets arrive after any of its successors, thus we also call them *late* packets.

In the context of the present work, it is also important to consider packets that arrive sooner than expected. We call them *early*, or *premature* packets. We say a packet is *early* if, at the time of its arrival, its sequence number is greater than $max_sn + 1$. For instance, if the downstream receives the packet sequence $\{1, 2, 4, 5, 3\}$, we only consider packet 3 to be reordered, and say that 4 is premature.

We can also calculate the *displacement* of any packet at the time of its arrival as $displacement = (max_sn + 1) - received_sn$. With this definition, late packets will have a positive displacement, and premature packets will have a negative displacement. For instance, in the above packet sequence, packets 1, 2 and 5 will have a displacement equal to zero, packet 4 will have a displacement of $-1$, and packet 3 will have a displacement equal to 3.

## 3 Related Work

The literature on network monitoring is extensive and covers many solutions that follow different approaches. We can classify existing solutions as *Host-Based*, *Switch Assisted* and *In-Switch*, according to the location where the monitoring data is collected and processed.

Host-Based solutions monitor the network solely from the end-hosts, and often rely on the exchange of additional packets to perform monitoring tasks. For example, PingMesh [1] is able to detect abnormal latency in data center networks by making end-hosts *ping* other network nodes and measure the elapsed time. NetBouncer [9], in turn, uses IP-in-IP to

measure the drop rates in multiple pre-defined paths, and correlates that data to locate faulty links in the network. Although these solutions are easier to deploy in already functioning networks, they are unable to collect crucial network insights, such as packet traces or the traffic intensity distribution. Additionally, the solutions that need to inject additional traffic in the network (we call them *active* solutions) tend to incur in unacceptable traffic overhead that often damages the network performance.

Switch Assisted solutions use switches to collect useful network insights, while using external components to process them. For instance, NetSight [2] and EverFlow [12] mirror application traffic to end-hosts, allowing to collect packet traces that enable anomaly detection and localization. Planck [6], in turn, mirrors the processed traffic to dedicated servers to detect real-time throughput and link congestion. Other solutions, such as OpenSketch [10] and UnivMon [4], leverage the Data Plane programmability to compute *sketches* that are then used by end-hosts to detect the targeted anomalies. The fine-grained information collected by these solution allows to detect a wider variety of anomalies, and to often detect their origin. Nonetheless, this approach requires switches to send the collected data to the external components, which may degrade network performance.

In-Switch solutions are able to execute the full anomaly detection logic in the Data Plane. HashPipe [8], for example, is able to perform heavy hitter detection entirely in the Data Plane. NetSeer [11], in turn, performs an inter-switch coordination algorithm to detect multiple anomalies, including per-flow packet loss. By detecting network anomalies in the Data Plane, one can drastically reduce the associated traffic overhead. Note that although these solutions may store the collected information in external devices for further analysis, they do not depend on them to monitor the network and to detect anomalies.

Our goal is to develop a Passive, In-Switch monitoring solution that relaxes the assumptions taken by NetSeer. While NetSeer requires monitored links to be physically connected to programmable switches, we aim to develop a solution able to detect packet drops between programmable switches that are connected by other non-programmable switches, or even an external network, that may reorder, duplicate, and drop packets. To fulfill our objective, we propose an inter-switch drop detection algorithm that extends the one used in NetSeer by tolerating packet reordering. In addition, we adapt the Failure Inference algorithm used in NetBouncer [9] to help pinpoint faulty links.

## 4 Window Based Monitoring

This section introduces Window Based Monitoring (WBMon), a system that detects packet drops in the network and identifies the lossy links that caused them. It leverages the availability of programmable switches (we call them *m-switches*) to insert observation points inside the network, but improves over state-of-the-art solutions (namely, NetSeer and NetBouncer) in several aspects:

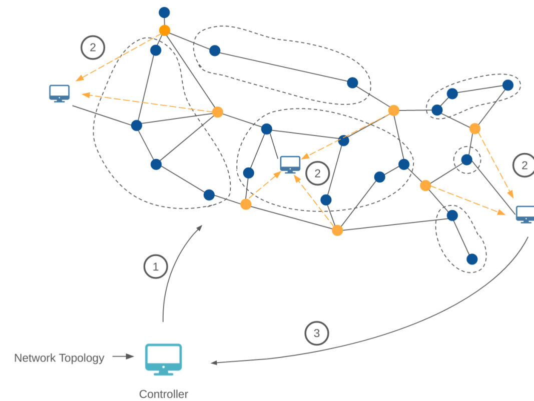



**Figure 1.** WBMon workflow overview

First, NetSeer requires every monitored link to be physically connected to two m-switches, which greatly increases the cost of deploying that solution in legacy networks. To do so, operators must either select a small subset of links to be monitored, which hinders solution coverage, or upgrade a larger amount of switches, which comes with a higher cost. Instead, WBMon is able to detect packet drops in arbitrary long sequences of links that are delimited by m-switches. This way, we can cover a wider range of links, while keeping the number of monitoring switches at a minimum. Additionally, it allows to gradually deploy and increase the solution coverage and accuracy by upgrading new switches over time.

Second, by using passive monitoring, our solution is able to directly monitor application traffic. This not only reduces the signaling overhead required by active monitoring solutions, but also allows to detect network failures that only affect application traffic, such as routing blackholes or ACL misconfigurations. Additionally, passive monitoring allows to collect data at higher rates with no additional cost, which allows to detect transient faults and presents a significant improvement over NetBouncer [9].

Finally, by monitoring packet drops across well defined paths in the network, WBMon is able to identify the set of links that *may* have caused each drop. We incorporated NetBouncer's Failure Inference Algorithm [9] to correlate the packet drops detected along multiple paths and *identify* the faulty links in the network. However, NetBouncer is restricted to monitor the network from its end-hosts, which limits the variety of paths that can be monitored. By performing in-switch monitoring, WBMon can monitor a wider variety of paths and to collect more and better data, which improves the failure inference accuracy.

### 4.1 System Architecture Overview

Figure 1 presents a diagram of WBMon's workflow. We can describe it as follows: Before deploying WBMon, network operators must determine the optimal location for the m-switches, given the target network topology (① in Figure 1).

The m-switch placement must be such that the network links are *identifiable* [9], *i.e.* the data collected in the monitored paths should be enough to identify the lossy links in the network.

Finding the best location for m-switches is an example of an NP-Hard problem, called the facility location problem. Panopticon [3] provides an efficient algorithm to determine the legacy switches that should be upgraded first, according to the operators needs. For this reason, solving this problem is out of the scope of the current work. Nonetheless, Section 5 discusses the implementability of WBMon in different network topologies.

After having the m-switches deployed in the network (the yellow circles in Figure 1), the Controller configures them with the forwarding rules required to monitor the network. This configuration can lead the m-switches to use different routes for different flows, in order to increase the number of monitored paths. Then, as the application traffic circulates through the network, each pair of m-switches executes the coordination algorithm described in Section 4.3 to count the number of packet drops that took place in each of the paths connecting them. The m-switches regularly send the number of detected packet drops and the number of processed packets to the network *Analysers* (② in Figure 1), which in turn run the Failure Inference Algorithm to locate the lossy links in the network and report them to the Controller (③ in Figure 1). Although the default behavior is waiting for the information to arrive from the m-switches, Analysers can also query it if necessary.

When defining the m-switch placement, the Controller can also split the network into multiple *neighborhoods* (Section 4.2), depicted by the dashed regions in Figure 1. This technique reduces the task of locating the faulty links in the entire network to locate the faulty links in each individual neighborhood. Additionally, by assigning each Analyser to subsets of neighborhoods, we allow WBMon to scale with the network size.

### 4.2 Underlying Model

WBMon considers a network consisting of a mix of programmable and non-programmable switches connected by bidirectional links, where only the programmable switches have monitoring capabilities. This network is modeled as an undirected graph $\mathcal{G} = (\mathcal{S}, \mathcal{L})$, where $\mathcal{S}$ denotes the switches, and $\mathcal{L}$ the set of bidirectional links. Additionally, $\mathcal{S}$ is partitioned into two subsets $\mathcal{M}$ and $\mathcal{R}$, where the former contains all the programmable switches (*monitoring switches*, or *m-switches*), and the latter the non-programmable switches (*regular switches*, or *r-switches*). Each link $l_i \in \mathcal{L}$ has a certain probability of successfully transmitting a packet, denoted as $x_i$. We assume that the success probabilities of different links are independent [9]. We also consider that both the topology and the forwarding rules at each switch are known, and do not change over time. This allows us to define a *virtual connection*, or *vconn*, as a sequence of links that connects two m-switches, without containing any loop and without passing through any m-switch. We assume that a vconn may drop, reorder and duplicate packets, and denote the set of all vconns in $\mathcal{G}$ as $\mathcal{V}$. Note that, as opposed to links, vconns are unidirectional. If two links participate in the same set of virtual connections, we say those links are *indistinguishable* since it is impossible to distinguish them using only the data collected by the m-switches. We denote the set of links that cannot be distinguished from $l$ as $\mathcal{I}(l)$. This concept will be relevant in Sections 4.4 and 5.

We also define a *neighborhood* as the set of r-switches and links that form a connected component on the network obtained after removing from $\mathcal{G}$ the m-switches and the links that directly connect two m-switches [3]. This notion is useful since it allows to analyze different neighborhoods independently, which simplifies the problem we are trying to solve, and allows our solution to scale to larger networks.

### 4.3 Drop Detection Algorithm

To better understand the drop detection algorithm, let's assume we have an infinite buffer split into different slots with $W$ bits each. Let's also consider that the arriving packets carry a sequence number (*sn*), corresponding to the order in which they were sent. During the rest of the document, we will refer to the slot in position $i$ as $s_i$, and to packet with sequence number $x$ as $p_x$. We call *current slot* to the slot the arriving packet belongs to, and *newest slot* to the slot that received the packet with the highest sequence number.

The buffer is initially filled with zeros and whenever a packet arrives, the m-switch writes a 1 in the position corresponding to the packet's sequence number. That position consists of a pair (*slot_idx*, *offset*), where the first field determines which slot will register that packet, and the second determines the bit that will be set to 1 in that slot. These values are calculated as follows:

$$slot_idx = \left\lfloor \frac{sn}{W} \right\rfloor ; offset = sn \bmod W$$

Since packets may be reordered, reporting a packet as dropped immediately after receiving one of its successors could be premature, since that packet may be late. To avoid premature reports, we define a tolerance window that waits for late packets to arrive before reporting them as lost. This window comprises the *tws* slots that record the packets with the highest sequence numbers. More precisely, the tolerance window ranges from $s_{newest_slot-tws+1}$ to $s_{newest_slot}$, inclusive. We call them "tolerance slots", since they are used to tolerate packet reordering.

As new packets arrive, the newest slot will eventually be updated and consequently the tolerance window will slide over the buffer. We can determine the number of dropped packets by counting the number of remaining zeros in the slots that exit the tolerance window. Note that despite using a tolerance window, this mechanism will still produce false positives if $p_i$ arrives after $p_{i+W\times tws}$, which stresses the importance of having an adequate *tws*:

In the current Tofino implementation, we stored the value of each slot in a Register. However, the finite resources available in hardware forced us to reuse the same Register for multiple slots. We assume that the m-switch contains $N$ available Registers. Registers were reused in a round robin basis, i.e., slots $\{0, N, 2N, ...\}$ are assigned to Register 0, slots $\{1, N+1, 2N+1, ...\}$ are assigned to Register 1, and so on. We say that slots stored in the same register are *cohabitants*, and that a slot is *active* if it contains information about which packets have already arrived, and *inactive* otherwise.

To assure correctness, each Register must be cleaned before being reused by another slot, otherwise the remaining bits set to one could conceal some packet drops. For this reason, we introduced a cleaning window that is responsible to *clean* the *dirty* slots that exit the tolerance window range as it advances. We call the slot that is being cleaned the *cleaning slot*, or the *slot to clean*. The cleaning window comprises at most *cws* slots and is, by definition, mutually exclusive with the tolerance window. To maximize reordering tolerance, we set the window sizes such that $cws + tws = N$. Note that if there are no dirty slots, the cleaning window has a size of zero. The current implementation is able to clean an entire slot for each processed packet. Hence, we conclude that the cleaning window advances $W$ times faster than the tolerance window.

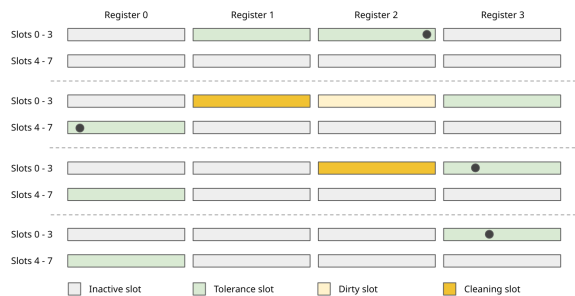



**Figure 2.** Tolerance Window update

Figure 2 illustrates the process of advancing the tolerance window and cleaning the dirty slots during the arrival of four packets, in an m-switch with $N = 4$ Registers. In this diagram, slots are represented by rectangles and are arranged such that cohabitant slots are displayed in the same column. The gray circle indicates the position where each packet will be registered.

Initially, we assume the buffer is in its normal state, having no dirty slots and receiving a packet belonging to the tolerance window. In this example, the tolerance window consists of $s_1$ and $s_2$. Let's suppose that the second packet is an early packet, belonging to $s_4$. Since the previously stored *newest slot* was $s_2$, its arrival will cause the tolerance window to advance by $4 - 2 = 2$ slots. Consequently, $s_1$ and $s_2$ become dirty slots and must be cleaned, so that $s_5$ and $s_6$ are ready to receive new packets (note that $s_1$ and $s_5$, and $s_2$ and $s_6$ are cohabitants). During the arrival of the second packet, we can immediately clean $s_1$ and detect which of its packets were lost by counting the remaining zeros in that slot. Similarly, we can do the same to $s_2$ when the third packet arrives. Finally, when the fourth packet arrives, there are no more dirty slots and the buffer returns to its normal operation.

The algorithm described in this section is trivially extended to a network with multiple vconns. To do so, each m-switch must keep, for each incoming vconn, $N$ Register arrays to store the buffer, and two Register arrays to store the *newest slot* and the *slot to clean*. For each outgoing vconn, each m-switch must only keep a single Register array, that tracks the next sequence number that is going to be sent to the respective vconn. In this work we assume that the Controller regularly resets the buffers and sequence numbers of m-switches such that sequence numbers do not grow boundlessly.

The resources available in the adopted switch model allowed us to use $N = 4$ Registers with $W = 32$ bits each, for 256 vconns. We believe that, as these switches evolve, there will be more Registers available and that, in time, these buffers may be larger and comprise more slots.

#### 4.3.1 Solution Correctness.

From Figure 2, one may notice that the arrival of packets with certain sequence numbers may lead our solution to produce incorrect results. For instance, if a packet arrives after its slot is cleaned, we know that packet was falsely considered dropped. We can define a safety slot interval that guarantees the solution correctness if every packet falls in it. This interval can be described with the following equations:

$$current\ slot > slot\ to\ clean \tag{1}$$

$$current\ slot < slot\ to\ clean + N \tag{2}$$

Where *current slot* denotes the slot the arriving packet belongs to, and *slot to clean* denotes the slot that will be cleaned during the processing of that packet. If there are no dirty slots, the *slot to clean* corresponds to the first tolerance window slot.

The above integrity conditions may not hold in a real-world scenario, thus it is important to determine the switch behavior when these conditions are not met. Ideally, m-switches would be able to detect the violation of these conditions, and emit a message reporting the incident. However, the hardware limitations of the Tofino switches inhibited this behavior. We concluded that, when receiving a late packet that violates condition 1, the best response is to not register that packet in the buffer and to leave the number of detected drops intact. This is because the late packet may be a duplicate, and doing so would lead our solution to produce False Negatives. When receiving an early packet that violates condition 2, the best action to take is to proceed normally, *i.e.* update the newest slot and register the incoming packet in the respective slot. We argue this is the best action to take since not advancing the tolerance window would completely stop WBMon in the presence of a large drop burst.

As discussed in Section 2, it is impossible to simultaneously clean and update a Tofino Register, hence it is important

to decide which action to take if the cleaning slot cohabits with the current slot. In the current implementation, we opt to clean instead of updating the Register, since doing so will produce at most one False Positive, while the alternative would produce at most $W$ False Negatives.

### 4.4 Failure Inference Algorithm

NetBouncer [9] proposes an optimization algorithm to locate the faulty links in a network, which we will briefly describe, for self-containment. For a more in-depth analysis, the reader can consult [9].

By assuming that the packet loss events are independent in different links, one can calculate the success probability $\hat{y}_i$ of $path_i$ as the product of the success probabilities of the links composing it. More precisely:

$$\hat{y}_i = \prod_{j:l_j \in path_i} x_j$$

One can combine this notion with the measured success probabilities of each vconn ($y_i$), and formulate an optimization problem that finds the values of $x_i$ that minimize the error between $y_i$ and $\hat{y}_i$:

$$\begin{aligned} \text{minimize} \quad & E = \sum_{j:vconn_j \in \mathcal{V}} (y_j - \prod_{i:l_i \in vconn_j} x_i)^2 \\ \text{subject to} \quad & 0 \le x_i \le 1, \forall i \end{aligned} \tag{3}$$

After finding the optimal values of $x_i$, NetBouncer identifies the faulty links as the ones that have a success rate lower than a user defined threshold.

Note that NetBouncer calculates $y_i$ as the ratio of probes that returned to the sending host. Instead, our solution calculates those values using the drop and packet counters from the m-switches.

Since indistinguishable links, by definition, belong to the same set of vconns, permuting the $x_i$ of those links will not change the value of $E$, thus it is possible that the Failure Inference algorithm blames healthy links that could not be distinguished from faulty links. For this reason, the WBMon Analysers yield the estimated success probabilities of each link along with the sets of indistinguishable links. With this information, network operators should investigate both the links with values of $x_i$ below the stipulated threshold, as well as the links that cannot be distinguished from those. We argue that under a link identifiable [9] m-switch placement, there will be no indistinguishable links.

## 5 Evaluation

This section details the experiments employed to evaluate the developed solution. We start by measuring the Drop Detection algorithm accuracy for different packet drop and reordering configurations, and discussing its limitations. Then, we perform a theoretical analysis that aims to understand the impact of the m-switch arrangement on WBMon's performance, and study the viability of this solution in two common real world topologies.

### 5.1 Drop Detection Algorithm

To evaluate the Drop Detection algorithm we developed controlled benchmarks to understand to which extent it tolerates packet loss reordering, and then test it against realistic packet traces. These experiments were executed using a single *m-switch* and a single Controller, to reduce possible errors originated outside of our algorithm. Both the Controller and the *m-switch* were executed on a Virtual Machine running Ubuntu 20.04.4 LTS, with 4GB RAM. The *m-switch* was emulated using the Intel SDE 9.7.0. The used implementation contained a tolerance window of size $tws = 2$, and was configured to use $N = 4$ registers of $W = 32$ bits each.

In each experiment, we used a traffic generator to produce packet sequences containing arbitrary packet drops and reordered packets. Packets were guaranteed to arrive the switch in the same order they were generated. After executing each experiment we queried the number of drops detected by the switch (*detected_drops*). By comparing it with the number of actual drops (*n_drops*), we can calculate the number of False Positives ($\#FP = \max(0, detected_drops - n_drops)$), and False Negatives ($\#FN = \max(0, detected_drops - n_drops)$).

**5.1.1 Reordering Tolerance.** In this experiment we generated multiple traces, each consisting of 266 packets starting at the same sequence number. In every trace we reordered $p_{165}$ and varied the displacement from $-NW$ to $NW$. Additionally, the switch was reset to the same initial state before sending each trace. As the traces used in this experiment did not drop any packet, we will only analyse the number of generated False Positives.

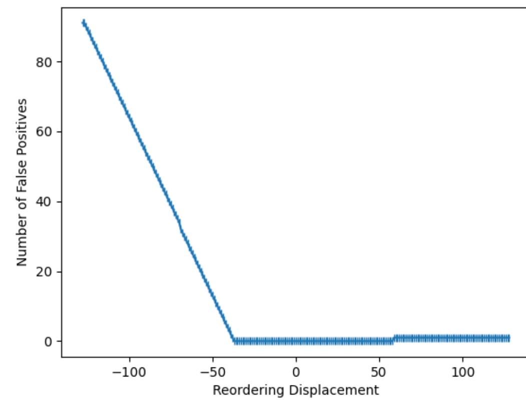



**Figure 3.** Number of False Positives (*#FP*) for different Reordering Displacement values

The results shown in Figure 3 reveal that WBMon produced no False Positives for traces with a reordering displacement in range $[-37, 59]$; a single False Positive for traces with a reordering displacement greater than 58; and a linearly decreasing number of False Positives for traces with reordering displacement lower than -37.

We can understand that the traces with reordering displacement greater than 59 generated a single FP since in those traces, the reordered packet arrived *after* its respective slot was cleaned, thus it was considered dropped before it arrived to the switch. The traces with reordering displacement lower than -37, in turn, cause the reordered packet to arrive too early, inducing a premature tolerance window update. When this happens, the switch will start cleaning the generated dirty slots before all of its packets have the chance to arrive. Note that the earlier a packet arrives (the smaller the reordering displacement), the more of its predecessors will be falsely considered dropped.

**5.1.2 Drop Burst Tolerance.** In this experiment we generated several packet traces that would drop a single burst of successive packets. The burst size varied from 0 to $3WN-1 = 383$. To isolate the effect of the drop burst, these traces were deprived from packet reordering and duplication. Additionally, to avoid external noise, every trace started with the same sequence number (32) and the drop bursts also started in the same packet ($p_{112}$).

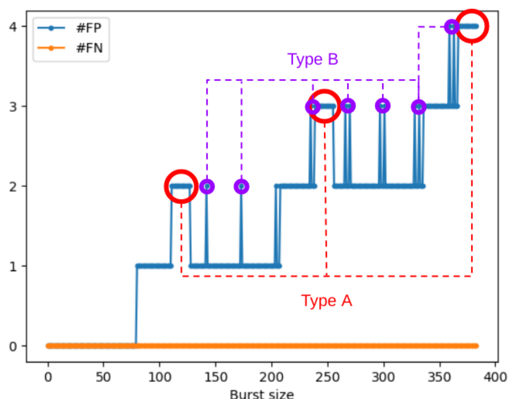



**Figure 4.** Number of False Positives and False Negatives generated for different drop burst sizes

The results shown in Figure 4 reveal that WBMon is able to correctly identify every packet drop, despite of the burst size. However, as the burst size increases, our solution starts producing False Positives. This happens since, for drop bursts larger than the buffer size, the number of dirty slots will be larger than $N$. When this happens, as the packets arrive after the drop burst, it is inevitable that some of them will cohabit the cleaning slot, and thus will not be registered. With bigger drop bursts, the more packets will be affected by this situation. We can also note small False Positive spikes, highlighted by the purple and red circles in Figure 4. The FP of Type A are generated when the drop burst has a size such that the first packet arriving after the burst is registered in a slot whose bit of the corresponding offset is already set to one. The FP of Type B, in turn, occurs when one of the packets that could not be registered belongs to offset $W-1$. In this situation, its successor will belong to offset 0 of the consecutive slot, coinciding again with the cleaning slot.which will cohabit the next cleaning slot and consequently generate an additional FP.

Note that our solution may generate False Positives in the presence of drop bursts larger than $(W-(W-1)) + W \times (tws + 0 \times N) = W \times tws + 1$ In the used implementation that represents less than $\frac{2}{65} = 3\%$ of the dropped packets, in the worst case.

**5.1.3 Performance with realistic traces.** To predict WBMon's performance under realistic scenarios, we generated multiple traces where we varied the drop and reorder probabilities such that they resemble a realistic scenario. The reordered packet displacement followed a normal distribution $\mathcal{N}(\mu = 0, \sigma^2 = 0.75^2)$ [7]. The reorder and drop ratios used for each scenario are summarized in Table 1. In each scenario, we calculated the False Positive Rate (FPR) and the Detected Drop Rate as follows:

$$\text{FPR} = \frac{\#\text{FP}}{\#\text{packets}}; \text{Detected Drop Rate} = \frac{\#\text{drops}}{\#\text{drops} + \#\text{packets}}$$

| Scenario | Configuration | | FPR | Detected Drop Rate |
|---|---|---|---|---|
| | Reordering Rate | Drop Rate | | |
| Normal Operation | 1.65% | 1% | 0.00 | 1.00% |
| Failed Link | 1.65% | 90% | 0.05 | 90.00% |
| | 1.65% | 95% | 2.42 | 95.24% |
| Faulty Switch | 70% | 5% | 0.00 | 5.02% |
| | 75% | 15% | 0.00 | 15.02% |
| DoS Attack | 70% | 80% | 0.00 | 80.01% |
| | 60% | 90% | 0.18 | 90.03% |
| | 70% | 90% | 0.19 | 89.96% |

**Table 1.** Detected Drop Rate under different realistic scenarios

Table 1 presents the FPR and Detected Drop Rate generated in each experiment. The measurement values shown in each column correspond to the average of 10 executions. This experiment produced no False Negatives in every execution, which indicates that our solution is able to effectively detect every packet drop that takes place in the monitored vconn.

We can observe that WBMon produces no False Positives in scenarios with drop rates lower or equal to 15%, despite of the amount of reordered packets. However, when the drop rate is 90%, and the reordered packet ratio is 1.65%, our solution produces an FPR of about 0.05‰. If the number of reordered packets increases to 60% and 70%, while keeping the amount of reordered packets, the value of the FPR increases to 0.18‰ and 0.19‰, respectively. This result suggests that under higher levels of packet loss, our system is more sensitive to the amount of packet reordering. When the monitored vconn drops 95% of its traffic, there is a great increase in the number of False Positives generated by our solution. Nonetheless, the amount of False Positives is relatively small when compared to the number of packets processed by the switch. Under these circumstances, WBMon has an FPR of 2.42‰.

These results indicate that our solution is able to correctly detect the drop rate of the monitored virtual connections. In most scenarios, the difference between the real and estimated drop rates was lower than 0.05%. The scenario that simulated a faulty link with a drop rate of 95% was the one in which the estimated drop rate varied the most. In this scenario, the expected drop ratio was 95.24%.

**5.1.4 Convergence Time.** This experiment aimed to estimate the time required for the Drop Detection Algorithm to converge to the real drop rate of each monitored vconn. To do so, we simulated the emission of several packets through a vconn with a drop rate of $y$. For each packet, we generated a random number and compared it with $y$ to determine if the packet was dropped. We obtained the estimated drop rate $\hat{y}$ of that vconn as the percentage of sent packets that were dropped. We counted the number of packets we had to send such that $|\hat{y} - y| < \epsilon$. With this value, the detection time is trivially calculated by dividing the number of sent packets by the vconn throughput. We assumed that the average packet size is 1KB [11].

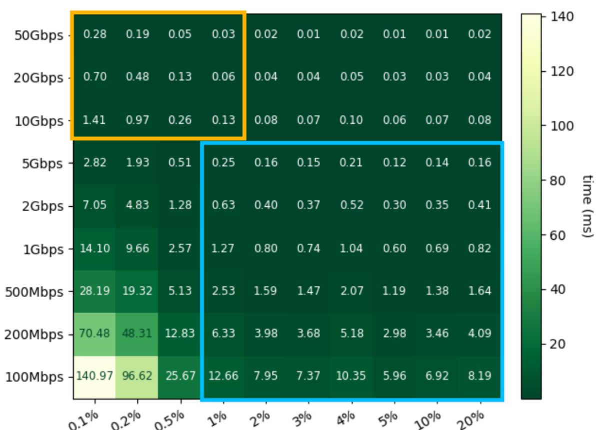



**Figure 5.** Time (ms) required to approximate each vconn drop rate for different vconn loss rates ($x$ axis) throughput values ($y$ axis), with $\epsilon < 0.05\%$

Figure 5 shows the time that the Drop Detection Algorithm takes to correctly approximate the vconn drop ratio with an error lower than $\epsilon = 0.05\%$, for different loss rates (x axis) and throughput values (y axis). The estimated times for ISP and data center networks are highlighted by the blue and orange rectangles, respectively.

The results show that WBMon takes less than 13ms to detect the vconn loss ratio for ISP networks. Moreover, if the traffic throughput in ISP networks is higher than 1Gbps, it takes less than 2ms to do so. In data center networks, in turn, the detection time for WBMon is lower than 2ms. This means our solution is able to detect transient failures of at least 2ms, which 5 orders of magnitude faster than active approaches [1, 9].

### 5.2 M-Switch Placement

In the following sections we study different m-switch arrangements on two real-world topologies, and evaluate their impact on WBMon's performance, more specifically, on the amount of faulty links successfully and falsely identified. We start by describing the followed procedure in Section 5.2.1 and then discuss the results obtained in Sections 5.2.2 and 5.2.3.

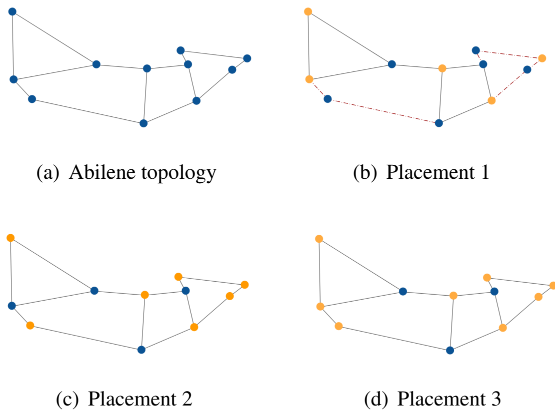
**5.2.1 Simulation Setup.** For each analysed topology, we select several m-switch arrangements and select a random subset of links $\mathcal{F} \subseteq \mathcal{L}$ to be faulty. Then, we assign a success probability $x_i$ to each, according to the loss model used in [9]: We then generated the drop reports that Analysers would receive from the m-switches under the given configuration, according to the procedure followed in [9]. Finally we fed the drop reports of each vconn to the Failure Inference algorithm to obtain the set of "blamed" [9] links $\mathcal{B}$. In this experiment we blamed all the links with the estimated success probability $\hat{x}_i$ lower than 99.8%, as well as all the other links that could not be distinguished from those.

By comparing the faulty and blamed links, we were able to calculate the number of True Positives ($\#TP$), the Recall ($\#TP/\#\mathcal{F}$) and Precision ($\#TP/\#\mathcal{B}$) of the failure inference algorithm.

**5.2.2 ISP Topology.** This experiment was conducted on Abilene, a real-world ISP topology extracted from the Internet Topology Zoo. We started by generating several random m-switch placements, while varying the percentage of monitoring switches in the network between 40% and 80%. We generated 100 random m-switch placements for each m-switch ratio, and for each placement and faulty link ratio, we selected 100 subsets of faulty links. This allowed us to study how the amount of m-switches affects WBMon's ability to locate faulty links. Then, we repeated the same experiment with three hand-made m-switch arrangements that aimed to reduce the number of indistinguishable links, while using the minimum amount of m-switches. Figure 6(a) depicts the Abilene Topology, and Figures 6(b) to 6(d) illustrate the hand-made arrangements used in this experiment. r-switches are depicted by the blue circles, and m-switches by the yellow circles.

The results obtained in this experiment are summarized in Figure 7. The graphics are organized as a grid, where the first row contains the Recall scores for the different configurations, and the second row contains the Precision scores. The graphics on the left are relative to the random m-switch arrangements, and the graphics on the right display the results of Placement 1 (P1), Placement 2 (P2), and Placement 3 (P3). For simplicity, we will refer to the random m-switch arrangement with $X\%$ of m-switches as $R(X)$.

These results indicate that WBMon is able to detect 70% of the faulty links (Recall), while monitoring the network from only 40% of its switches; and 80% of the faulty links while monitoring the network from 60% of its switches. The Recall scores consistently increase with the m-switch ratio,



(a) Abilene topology (b) Placement 1

(c) Placement 2 (d) Placement 3

**Figure 6.** Hand-made m-switch placements for the Abilene topology

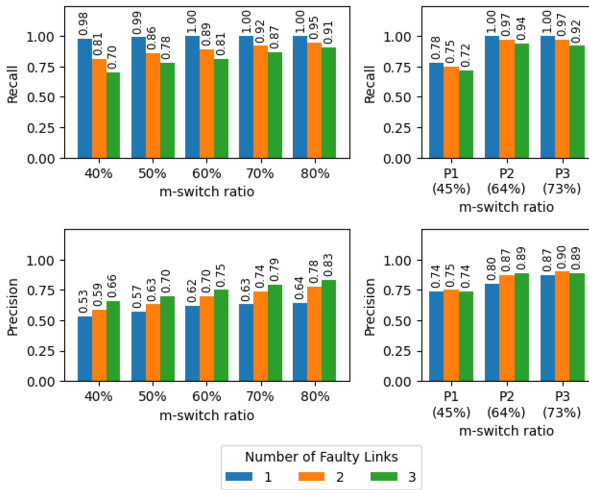


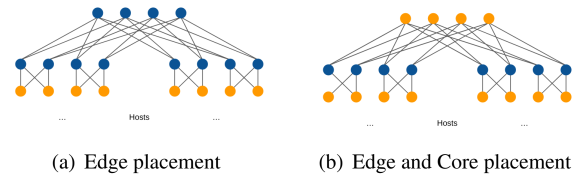

**Figure 7.** Recall and Precision of the Failure Inference Algorithm for the Abilene topology. Random m-switch placements on the left, Hand-made m-switch placements on the right.

indicating that adding more m-switches to the network increases the number of faulty links that can be successfully detected. Despite having relatively high Recall scores, the random m-switch placements for this topology lack in Precision, scoring values ranging from 53% to 83%. This indicates that WBMon often blames the wrong links in this topology. This result comes from the existence of indistinguishable link sets (ILS) originated by the random m-switch placements.

We can observe a significant increase in the Recall and Precision of WBMon when using the hand-made m-switch arrangements. Namely, P2 and P3 were able to correctly identify more faulty links (higher Recall) and blamed fewer healthy links (higher Precision) than *R(80)*, while using only 64% and 73% of the switches to monitor the network. Additionally, P1 obtained higher Precision scores than R(70) in the experiments with less than three faulty links, and higher Precision scores than R(50) in the experiments with 3 faulty links. These results stress the impact of having a good m-switch placement. In fact, having optimal m-switch arrangements reduces the amount of m-switches required to monitor the network with the same Precision and Recall.

**5.2.3 Data Center Topology.** Clos topologies were widely used in former data center networks [1, 9, 12], and their well defined structure makes them a good candidate to study different m-switch placements. In this experiment, we study a 4-ary Fat Tree topology, which is a special instance of a Clos network. The switches of Fat Tree topologies are divided into the Edge, Aggregation and Core layers: The hosts connect to the Edge layer, the Edge is connected to the Aggregation layer, and the latter is connected to the Core layer (Section 5.2.3). Since hosts are exclusively connected to the Edge layer, the traffic also follows a predictable pattern. For this reason, we tested this topology using two hand-made m-switch arrangements.

The first m-switch placement considered was obtained by upgrading only the switches at the Edge layer (Figure 8(a)), which resulted in a configuration with an m-switch ratio of 40% and no indistinguishable links. This placement corresponds to the minimal set of monitoring switches that allow to monitor the entire traffic circulating in the network. The second m-switch arrangement, in turn, consisted of updating the switches at the Edge and Core layers. This resulted in an arrangement with 60% of monitoring switches and no indistinguishable links. Note that the paths measured with the last configuration are the same as the ones monitored in NetBouncer's link identifiable probing plan [9].




(a) Edge placement (b) Edge and Core placement

**Figure 8.** Different m-switch placements for a 4-ary Fat Tree Topology

The results shown in Figure 9 reveal that the Edge placement is able to successfully identify 99% of the faulty links in the network, while using only 40% of monitoring switches. However, it produces too many False Positives, which makes the Precision of that arrangement have scores of at most 51%. This suggests that the Edge arrangement is not link identifiable, and that the number of monitoring points in this configuration was not enough to correctly monitor the network.

By adding the Core switches to the monitoring set, we were able to drastically improve the solution's Precision. In fact, the second m-switch configuration was able to correctly identify 100% of the faulty links and did not blame any non-lossy link, while using only 60% of monitoring switches.

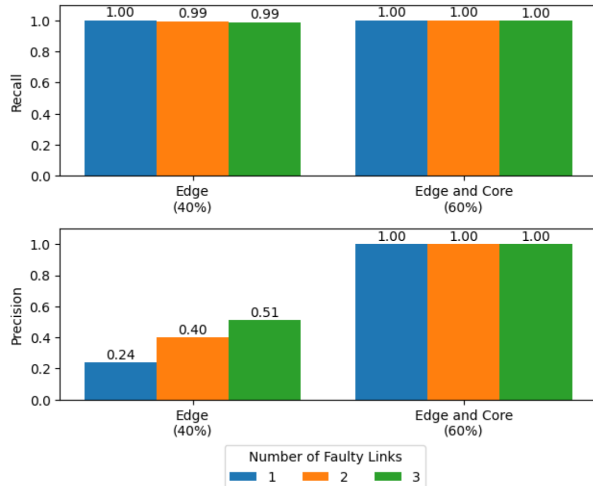



**Figure 9.** Comparing the Recall and Precision of the Edge and Edge+Core m-switch placements in the Fat Tree topology.

We can understand this result as this arrangement is link identifiable [9].

This experiment demonstrates that, for certain network topologies, it is possible to reduce the number of monitoring switches without damaging the monitoring quality.

# 6 Conclusion

Detecting packet loss and locating its origin is a crucial task to manage and enhance the performance of computer networks. In fact, multiple solutions aim to solve this problem with countless techniques. Some of them provide fine grained metrics that allow to locate faulty devices with high precision, at the cost of requiring multiple observation points. This requirement increases the solution deployment cost, which may prevent operators from adopting it. Others aim to reduce this cost by computing coarse grained statistics that may not be enough to precisely locate the faulty devices.

In this thesis, we developed and evaluated WBMon, a passive monitoring solution that sits in the middle ground between the existing solutions. On the one hand, we leverage the Data Plane programmability to perform an inter-switch coordination algorithm that allows to detect the exact number of dropped packets in individual network paths. On the other hand, by employing a correlation algorithm, our solution requires fewer observation points, which makes it cheaper to deploy in already functioning networks. Our evaluation demonstrates that the developed solution is able to correctly identify the majority of faulty links while requiring about half of the monitoring points.

# References


[1] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, London, United Kingdom, 139–152. https://doi.org/10.1145/2785956.2787496

[2] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Seattle, Washington, USA, 71–85.

[3] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. 2014. Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 333–345. https://www.usenix.org/conference/atc14/technical-sessions/presentation/levin

[4] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '16)*. Association for Computing Machinery, Florianopolis, Brazil, 101–114. https://doi.org/10.1145/2934872.2934906

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38 (mar 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[6] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*. Association for Computing Machinery, Chicago, Illinois, USA, 407–418. https://doi.org/10.1145/2619239.2626310

[7] Pedro Rodrigues Torres-Jr and Eduardo Parente Ribeiro. 2020. Packet Reordering Metrics to Enable Performance Comparison in IP-Networks. *Journal of Computer Networks and Communications* 1, 1 (2020), 1–8.

[8] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, Santa Clara, California, USA, 164–176. https://doi.org/10.1145/3050220.3063772

[9] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. Netbouncer: Active Device and Link Failure Localization in Data Center Networks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, Boston, Massachusetts, USA, 599–613.

[10] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th Usenix Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Lombard, Illinois, USA, 29–42.

[11] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. Flow Event Telemetry on Programmable Data Plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, Virtual Event, USA, 76–89. https://doi.org/10.1145/3387514.3406214

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