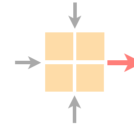
ETH zürich



Networked Systems
ETH Zürich — seit 2015

Retroactive Packet Sampling in P4

Master Thesis

Author: Clemens Klopfstein

Tutor: Georgia Fragkouli
Co-Tutor: Alexander Dietmüller

Supervisor: Prof. Dr. Laurent Vanbever

September 2023 to March 2024

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Retroactive Packet Sampling in P4

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Klopfstein	Clemens

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 25.03.2024	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Non-disclosure agreement protection

Parts of this thesis fall under the Non-disclosure agreement (NDA) set by Intel® and thus are either blacked out or omitted in this version. Please get in touch with the project's supervisor if you need more information.

Acknowledgments

I would like to thank Prof. Laurent Vanbever for offering the opportunity to write this thesis. Furthermore, this thesis would not have been possible without the critical questions and guidance of my advisors Dr. Georgia Fragkouli and Alexander Dietmüller. Additionally, I would like to thank Albert Gran Alcoz for the discussions on the Intel® Tofino™ as well as my friends for their support and satisfying distractions outside this work. Finally, I would like to thank my family for their support.

Abstract

To enable third parties to verify the performance of individual networks, Retroactive Packet Sampling (RPS) ensures that a network cannot bias the sample of traffic based on which a monitor is judging its overall performance. However, deploying RPS is challenging: it requires sampling primitives that are not available in traditional switches or are too complex for programmable switches.

This thesis implements RPS for the P4 software switch and the Intel® Tofino™, which requires grappling with the compute/memory constraints of programmable switches. Specifically, we explore two different implementations of RPS — one running entirely on the data plane and a hybrid one running on both the data and control plane — and evaluate the resulting tradeoffs. We find using a simulator that a Tofino can sample up to 61 Gbit/s (9.5 Mpps) of traffic capacity while minimizing additional bandwidth to the controller on a reference configuration and using 30% of the total data plane memory. With a more involved controller, a sampling capacity of up to 180 Gbit/s (28.28 Mpps) can be achieved without additional hardware.

Contents

Declearation of Originality 1

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Challenges 2
- 1.3 Contributions 2
- 1.4 Overview 2

2 Retroactive Packet Sampling 3
- 2.1 Analyzing network traffic 3
- 2.2 Packet Sampling 4
  - 2.2.1 Trajectory Sampling 4
  - 2.2.2 Delayed Disclosure 5
- 2.3 Retroactive Packet Sampling 5
  - 2.3.1 Resistance to attacks 6
- 2.4 RPS Algorithm 7
- 2.5 RPS Parameter Analysis 9
  - 2.5.1 Number of samples N 10
  - 2.5.2 Collection time T 11
  - 2.5.3 Quiet time κ 11
  - 2.5.4 Minimal rate r_{min} 12
  - 2.5.5 Selection probability σ 13
  - 2.5.6 Disclosure probability δ 13
  - 2.5.7 Tofino™ parameters 14
- 2.6 Reference Implementation 15
  - 2.6.1 Receipt 15
  - 2.6.2 Buffer management 16
  - 2.6.3 Expected performance 16

3 Programmable Data Planes 17
- 3.1 PISA 17
- 3.2 The P4 Language 18
  - 3.2.1 P4 Features 19
  - 3.2.2 Storage 19
  - 3.2.3 Traffic manipulation 21

4 Implementation **22**
4.1 Implementation Goals 22
4.1.1 Goals 22
4.1.2 Non-Goals 23
4.2 RPS on the Controller 23
4.2.1 Header mirroring 23
4.2.2 Digest mode 24
4.2.3 Interim conclusion about RPS on the controller 25
4.3 RPS on the Simple Switch 25
4.3.1 From SHA256 to CRC32 26
4.3.2 Buffer access in P4 28
4.3.3 Naive implementation 29
4.3.4 *Inverting* the RPS Algorithm 30
4.4 RPS on the Intel® Tofino™ 34
4.4.1 Stages 34
4.4.2 Storage 34
4.4.3 Controller communication 35
4.4.4 Timestamps 36
4.4.5 Final implementation 37
4.4.6 Not considered 37

5 Evaluation **39**
5.1 Hardware resource usage 39
5.1.1 Methodology 39
5.1.2 Implementations 40
5.1.3 Results 41
5.2 Accuracy 42
5.2.1 Simulation Setup 43
5.2.2 Methodology 43
5.2.3 Implementations 46
5.2.4 Overview 46
5.2.5 P4 Reference model 47
5.2.6 Tofino model 50
5.3 Performance 51
5.3.1 Setup 51
5.3.2 Methodology 52
5.3.3 Implementation 52
5.3.4 Controller-based sampling on the Tofino™ 53

6 Conclusion and Outlook **55**
6.1 Future work 55
6.2 Conclusion 56

References **58**

A Average packet size based on CAIDA traces **I**

B Trace configuration **II**

C Evaluation of the ideal RPS Algorithm **III**
C.0.1 Evaluation III

Chapter 1

Introduction

1.1 Motivation

The Internet is a decentralised mesh of computer networks, where each network is operated by an Autonomous System (AS). The best-known operators of AS are Internet Service Providers (ISPs) that offer customers Internet access to connect to services over the Internet. These ISPs also provide Service Level Agreements (SLAs), in which they guarantee their customers specific performance metrics of the connection, such as the minimal packet delivery rate or maximum latency [1]. Without controlling both endpoints of a path spanning possibly over multiple ISPs through the Internet, verifying these SLAs is challenging, as ISPs can detect certain measurement packets such as ICMP [2], and treat them preferable, i.e. prioritise them to exaggerate their performance. Hence, a solution in which an ISP cannot cheat and the path endpoints do not need to be controlled by the same party to collect metrics is required. This leads to the need to generate samples on the network path, requiring the assistance of ISPs and posing the challenge of devising a monitoring scheme that cannot be cheated on while keeping the performance overhead minimal. The former follows from the fact that the ISP is self-attesting its performance, and some ISPs have been caught tampering with requests [3], whilst the latter is needed to remain cost-effective and scalable, incentivising ISPs to partake.

In 2019, *Nikolopoulos et al.* [4] proposed Retroactive Packet Sampling (RPS) to address this issue. By buffering and conditionally releasing small traffic artefacts (samples) containing packet fingerprints, the sampling fate of a packet is unknown at the arrival time. It is only determined by a future packet using a probabilistic determination method. Forcing a time between these two events hinders the ISP from learning the sampling fate early enough to prioritise the samples and thus prevents prioritisation attacks. To protect the collection system, the monitor, from fake samples generated by an ISP to manipulate the traffic metrics, samples are collected along the network path over multiple ISP. Culprits can be identified by comparing samples from different ISPs and leveraging their business relationships.

We want to evaluate the implications of running RPS in programmable data planes such as P4 [5]. If possible, ISPs can reprogram their existing programmable data plane devices, removing the need for additional hardware and thus simplifying the roll-out of RPS, improving the localisation of Internet performance issues and SLA verification, leading to improved network transparency.

1.2 Challenges

With P4 running on top of a switch-oriented hardware architecture, many features commonly present in high-level languages, such as unbounded loops or freely addressable and allocatable memory, are missing. This poses a significant challenge to the adaptation of RPS that heavily relies on iterating over its buffer. Furthermore, P4 devices such as the Intel® Tofino™ only offer limited persistent memory, enforcing hard bounds on the sampling capabilities.

1.3 Contributions

We adapt the sampling algorithm of RPS from a loop-based form with multiple possible buffer modifications per packet into a P4-compatible form, requiring a single buffer operation per packet, overcoming the unbounded loop constraints. Furthermore, we reduce the required bytes per packet artefact to optimise the in-data plane memory and evaluate a hybrid approach to increase buffer capacities. Using the results of our simulator, we show that the modified algorithm achieves the same sampling accuracies as the original algorithm. Evaluating the implementation using a simulation of the Tofino™ with parameters presented in the RPS publication, a maximal performance of 9.5 Mpps, equivalent to 61 Gbit/s with an average packet size of 800 bytes used in today's Internet, is achieved.

The main contributions of this work are:

- Introducing the changes needed and tradeoffs faced to adapt the RPS Algorithm from the paper to the P4-programmable Intel® Tofino™.
- Providing multiple RPS implementations in P4 with different characteristics, focusing on the memory bottleneck.
- Evaluating the accuracy and sampling performance of the provided implementations using a simulator.

1.4 Overview

In Chapter 2, we briefly present RPS with a focus on the effects of the parametrisation on the buffer size, as this will be later identified as a bottleneck. In Chapter 3, the *P4* [5] language is presented, and the core features set of programmable data planes is introduced. We then present multiple implementations of RPS in P4, moving from a naive implementation with limitations to a Intel® Tofino™ compatible implementation, documenting the design decisions. The proposed implementations are evaluated in Chapter 5 with respect to the hardware usage, the sampling accuracy using a simulator, and the performance on actual hardware. Finally, a discussion and outlook are presented in Chapter 6.

Chapter 2

Retroactive Packet Sampling

Retroactive Packet Sampling provides traffic samples in the form of small artefacts called *receipts* to a trusted monitor outside of the ISP's control to verify the ISP's delay and loss properties. The sampling algorithm running on the ISP's untrusted hardware uses a combination of buffering and hash functions with strong randomisation properties to reduce the ISP's impact of misbehaviour. With well-chosen parameters for the algorithm, the monitor can determine the desired metrics with pre-defined statistical accuracies solely based on the artefacts, called disclosures, sent to the monitor. RPS can detect misbehaving ISPs down to an inter-ISP link, with business incentives between ISPs leading to the exact determination of the culprit.

We first introduce the need for network traffic analysis, followed by introducing two sampling approaches. Afterwards, the general concept behind RPS, including the resistance to attacks, is described. This is followed by the introduction of the sampling algorithm (RPS Algorithm) itself and a discussion of the parameters is followed. The chapter concludes with a brief introduction to the reference implementation.

2.1 Analyzing network traffic

Analysing network traffic is important due to many factors. On one side, different services require different traffic properties, while on the other side, insights into the network help troubleshooting and capacity planning. Whilst RPS is mainly designed to enforce QoS properties such as SLAs, it can also be used for network troubleshooting.

Quality of Service (QoS) Quality of Service mechanisms help applications receive the necessary network resources and operate without degrading quality. Requirements for telephony over the Internet (VoIP) and video streaming differ, as humans are sensitive to minor disturbances in real-time voice communication, while video players can fetch content ahead of time and buffer it.

Network troubleshooting Tracking errors in large networks is complex [6]. Network analytics help to find network issues by reporting real-time information about the network state. Seeing a packet's trace through the network drastically speeds up the localisation of misconfiguration and hardware faults.

Capacity planning With extensive networks and different traffic forwarding rules, tracking the network's overall state is non-trivial [6]. Traffic might be split to better utilise the available

bandwidth or cost-optimised paths may be chosen. In the case of link failures, alternative paths need to be selected, and after resolving the issues, traffic needs to be reallocated to the link. Analysing the network traffic helps find underutilised links and detect bottlenecks.

2.2 Packet Sampling

Computer networks often consist of many devices interconnected by various topology patterns. Some exist solely as local area networks, for example, in a data centre, while others span continents and consist of multiple self-regulated sub-networks such as the Internet. Resulting in complex configurations hard to understand [6]. In addition to the complex state, traffic rates in the hundreds of gigabits per second generally overwhelm monitors that determine traffic metrics [7]. Over the past decades, network researchers and engineers have implemented various algorithms that only look at traffic subsets by sampling packets. These algorithms are tailored to specific traffic metrics and patterns and shed insights into the network operations. With sampling only a subset of the packets, overall monitor resources can be efficiently used, and large amounts of traffic can be analysed.

But with only a subset being sampled, two questions are opened:

1. How can one decide if a packet should be sampled?
2. How can exploiting the sampling primitive be prohibited or reduced?

The following sections introduce two specific types of sampling relevant to this work.

2.2.1 Trajectory Sampling

In 2001, *Duffield et al.* [8] presented a sampling technique called *Trajectory Sampling*, used to determine if a packet should be sampled independently on the sampling node using hash functions. This results in a consistent sampling over the entire network.

Consistent sampling The packet must be sampled at all path nodes to derive statistics on a packet path or trajectory. This requirement is called consistent sampling. Consistent sampling defines the behaviour of a packet p being sampled on all observation points in the network, given that it is not dropped. Given two consecutive nodes x and y, then:

$$y \text{ samples consistently } := p \text{ sampled at } x \implies Pr[p \text{ sampled at } y | p \text{ received at y}] = 1 \quad (2.1)$$

Trajectory sampling algorithm A hash function over a constant packet part can consistently sample packets in a network Algorithm 1. The constant fields of network packets are all fields that are not modified by network routers and switches. Modifiable fields in a packet include the Time To Live (TTL) in the IP header and others. If the calculated hash value is below a certain threshold, set to determine the sampling percentage, the hash is sent to a monitor. If enough bytes are incorporated into the hash and the hash function has good randomisation properties, the packet hashes can be reliably used for packet identification over an entire network, as the probability of a different packet on a trajectory having the same hash is low, due to the randomisation properties leading to few hash collisions.

Duffield et al. [8] used a generic modulo operation to calculate the hash value, which was chosen according to the traffic intensity and length of the incorporated data. *Carle et al.* [9] used CRC and MD5 checksum to generate digest over the first 40 bytes modulo the immutable fields.

Algorithm 1 Possible Trajectory Sampling Algorithm

```
1: procedure TrajectorySampling(p : packet)
2:     if Hash(immutable(p)) ≤ Threshold then
3:         Send hash to the monitor
4:     end if
5: end procedure
```

Vulnerability to attacks Although the sampling algorithm is capable of consistent sampling and packet tracking over a network, a significant flaw is the predictability of the sampling decision. An ISP can determine if the packet is sampled at the time of its arrival at the device and decide to forward it on a faster path, i.e. prioritise it or use a path with less loss. From an outside observation, not controlling the sender and receiver, the delay and loss reported are better than the actual performance. Thus, an ideal sampling algorithm should make it complicated or expensive for a malicious ISP to lie about its performance.

2.2.2 Delayed Disclosure

A different sampling approach that aims to cheat on the performance metrics is called delayed disclosure, where samples are only determined after a certain time, assuming the network operator has already forwarded the packet. The main characteristic of this sampling type is using a buffer to store packets or packet artefacts for a certain period until a decision to sample a packet or its artefact from the buffer is made. This sampling approach has the advantage that a malicious actor cannot determine whether an arrived packet will be sampled, as this decision lies in the future. Examples of such algorithms are *Network Confessional* [10] or *PAAI-1* [11].

2.3 Retroactive Packet Sampling

Retroactive Packet Sampling(RPS) [4] is a sampling technique introduced in 2019 by *Nikolopoulos et al.* addressing the issues of packet tracking over multiple domains[1] (ISPs, AS) without trusting the operators of the sampling algorithm. Motivated by the desire to provide more network transparency, such as the information where packets are dropped or what delay is added, and detect traffic prioritisation, a combination of *consistent sampling* and *delayed disclosure* was used to create a new sampling algorithm. This work refers from now on to the publication [4] by *Nikolopoulos et al.* as *RPS paper*.

Sampling infrastructure Packets sampled at the ISP network edges by a sampling node as displayed in Figure 2.1. A small artefact called *receipt* is generated for every incoming packet. Based on the sampling scheme introduced later, a receipt is chosen as a sample, referred to as a *disclosure* and sent to the monitor. Using the consistent sampling properties, every sampling node creates the same disclosures - unless a packet is dropped. Based on these observations, the monitor determines the packet loss locality at a granularity of a link between two sampling nodes. The delay and loss can be determined on an arbitrary sub-path between sampling nodes using the obtained disclosures. All sampling nodes are untrusted as they estimate the performance of an ISP and thus are a likely target for modifications to exaggerate the performance metrics. The

[1] This work uses ISPs, ASs and domains interchangeably

monitor is trusted to not exaggerate performance metrics. In the ideal case, disclosures are made publicly available by an RPS framework and thus allow everybody to implement their monitor, reducing the need for trusting a third party. To infer additional information about packets, the first node not only sends a disclosure to the monitor but also sends metadata of the packet, such as the used protocols or ports, to the monitor. This additional information is used to index the network streams introduced later.

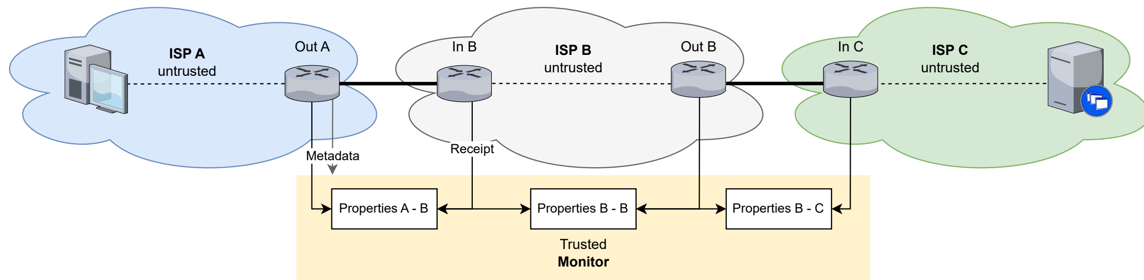


Figure 2.1: Setup for Retroactive Packet Sampling. Every ISP on the path provides the collected disclosures to the trusted monitor. The monitor can then infer the loss or latency of the flow between the client and server up to a granularity of an inter-sampling node link.

2.3.1 Resistance to attacks

With the sampling nodes running under the control of the evaluated ISPs, fake receipts or disclosures can be generated, and traffic can be prioritised. The former attack is detected and localised due to the consistent sampling properties of the sampling algorithm, leading to hurt business relationships between the ISPs. The latter attack is made expensive by the delayed disclosure properties of RPS.

Fake receipts The fake receipts attack can be split into multiple categories: adding fake receipts, modifying receipts and dropping receipts. The monitor catches all of these attacks:

Add A malicious ISP sends made-up receipts as disclosures to the monitor, hoping to influence the performance estimates. However, the consistent sampling primitives ensure that a packet is either sampled by everybody (except if it is lost) or by nobody. Therefore, the monitor will receive a superfluous disclosure and can reject it.

Modify When changing parts of a disclosure, such as the time of arrival, an ISP can pretend to have received a packet earlier and thus pretends to have a minor delay on the link than there truly is. However, due to the consistent sampling, the next node selects the same packet (receipt) as disclosure and sends the actual arrival time. The monitor determines the link delay, and due to the time shift, the perceived delay to the next node will be more significant than in reality. The monitor cannot determine which ISP is responsible for the increased delay so that it will blame both ISPs. However, both ISPs have access to the wire and can run telemetry on it; the second ISP can determine that there was no congestion or reason for additional delays and thus blame the malicious IPS, which damages the business relationships and incentivises the refrain of such actions.

Drop A malicious ISP can also drop disclosures instead of sending them to the monitor. However, there is no incentive to do this, as it will be held accountable for the loss, as the ISP on the

other side of the link will collect the disclosure.

Attacks could also be described for receipts, but the effects are not mentioned here as only a few are selected by the sampling scheme and receipts that are not selected do not impact the metrics the monitor collects.

Priorization attack A prioritisation attack occurs when a malicious ISP forwards "disclosure" packets over a link with lower latency or loss. As seen before, trajectory sampling is vulnerable to this attack. The RPS makes such attacks unlikely by enforcing receipts to be stored long enough before being disclosed. The malicious ISP in the attack scenario would wait until the disclosures are determined and then send them over the good path. With ISPs such as Cogent guaranteeing round trip times below 45 *ms* in North America [1], buffering receipts for 50 *ms* would make the attack infeasible as a link with latency -5 *ms* would be needed such that the packets arrive in due time at the next sampling node to honour the SLA. The proof that RPS achieves the attack resistance can be found in the *RPS paper*.

2.4 RPS Algorithm

The RPS algorithm determines the sampling fate of every packet that arrives at a node. Using hash functions with strong randomisation properties, an incoming packet is either chosen to be a direct disclosure that triggers the delayed disclosure of receipts from the buffer or the packet's artefact is stored as a receipt in the buffer. Delayed disclosures are receipts that are old enough and share properties with a direct disclosure.

Determining disclosures The RPS Algorithm (Algorithm 2) creates for every incoming packet an artefact called *Receipt*, consisting of a *flowid* that identifies the packet stream, a *timestamp* containing the arrival time of the packet and a *digest*, serving as a "packet fingerprint". This fingerprint is calculated using a hash function with strong randomisation properties over immutable packet parts to enforce consistent sampling. The sampling node has a fix-sized buffer where every created receipt is stored, with old receipts being overwritten by new ones. Based on the *digest* and the packet stream's rate, it is decided if the receipt is promoted to a direct disclosure (line 5). If not, the algorithm terminates; otherwise, the receipt is sent to the monitor (line 6), and the process of finding delayed disclosures is initiated (line 10). Iterating over the entire buffer, all receipts with the same *flowid* are removed and checked to be old enough (line 12) and it is checked if the hash of the buffered receipt and the direct disclosure fulfil a specific criterion (line 15). If both checks succeed, the packet is sent to the monitor as a direct disclosure (line 16).

Algorithm 2 Retroactive Packet Sampling Algorithm [4]

1: **procedure** RETROACTIVESAMPLING(p : packet)
2: $R' \leftarrow Receipt(p, currentTime)$
3: Add R' at the tail of the circular receipt buffer
4: $r \leftarrow packetRate(R'.flowID)'$
5: **if** $DiscHash(immutable(p)) \in DiscRange(r)$ **then**
6: Emit receipt R'.
7: **if** $LateDisclosure(flowID)$ **then**
8: Emit warning.
9: **end if**
10: **for** all Receipts R' in receipt buffer with $R.flowID = R'.flowid$ **do**
11: Remove R from the receive buffer.
12: **if** $(currentTime - R.time) \leq \kappa - \mu$ **then**
13: **continue**
14: **end if**
15: **if** $Hash(R.digest, R'.digest) \in Range$ **then**
16: Emit receipt R.
17: **end if**
18: **end for**
19: **end if**
20: **end procedure**

Disclosures and Receipts When arguing about receipts and disclosures, it is important to note, that they have the same data representation. There is no difference between them - the sampling algorithm sends a receipt to the monitor. However, to reason about the algorithm and its effects, the term direct disclosure is used to refer to a receipt being sent to the monitor on line 6 of the algorithm, while a delayed disclosure is a receipt that is sent to the monitor in line 16 of the RPS Algorithm.

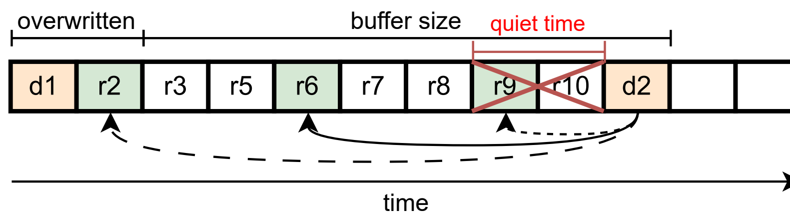


Figure 2.2: Buffer overview: disclosure $d1$ and receipt $r2$ have been overwritten, receipt $r6$ is picked by disclosure $d2$ as a delayed disclosure, while receipts $r9$ and $r10$ are too new to be considered. The receipts $r2$ and $r9$ would have been chosen as delayed disclosures if the buffer had been larger and the disclosure $d2$ had arrived later. The quiet time is the time to enforce the delayed disclosure properties.

Late disclosure With a circular buffer, old entries are overwritten to make space for new ones. Although the RPS Algorithm removes all receipts of a flow upon a direct disclosure, this might not happen early enough in some cases, and old receipts are dropped (Figure 2.2). As not sending a disclosure to the monitor indicates a packet loss, the algorithm informs the monitor with a late

disclosure warning if the previous disclosure is not in the buffer anymore. This warning points to the subsequent receipt in the buffer so that the monitor can determine the timespan of the missed receipt of the node.

Flows RPS determines the packet latency and the loss rate on the granularity of flows, which are identified by a *flowid*. Contrary to the TCP flows, identified by the 5-tuple, RPS flows are built upon IP address prefixes, combining the source and destination address prefixes to a flow identifier. As the flows do not allow for distinguishing HTTP and HTTPS traffic based on the port number, aggregates can be defined. A flow can be further split into smaller aggregates using the initial metadata sent to the monitor. Aggregates are not exclusive to a flow and can be defined across multiple flows. However, as introduced in Section 2.5, these aggregates might not fulfil the algorithm's criteria and thus require more time until enough samples are collected to determine the target metrics with statistical guarantees.

RPS does not limit the number of flows present at any given time,

2.5 RPS Parameter Analysis

So far, the RPS Algorithm was only introduced on a functional level without providing actual parameters such that the monitor can make statistically relevant claims. In this section, all parameters that control the algorithm are introduced and related to the size of the circular buffer β, as this will be a crucial challenge to overcome in the P4 implementations later presented in this thesis. Furthermore, the authors of the *RPS paper* designed the algorithm to use as little memory as possible since memory for buffering in networking is expensive. An operator defines the (γ, ε)-accuracy for the desired metric and the time T, in which enough samples should be collected to honour the statistical relevance of the claims made. Additionally, the operator specifies the quiet time κ, which must be chosen to reduce prioritisation attacks. Further parameters are the minimal rate r_{min}, which every flow must hold. Lastly, the probability of a direct disclosure δ and the selection probability σ are introduced.

The analysis shows the strong impact of the parameter choice of r_{min} and σ regarding the required buffer.

Parameter localisation As a sampling node is placed at each side of an inter-ISP link, the parameters must be chosen to satisfy the properties of this individual link's traffic. Hence, an ISP has to run an instance of RPS at every incoming and outgoing link, i.e. at every border router. The tracking of flows over multiple ISPs by the monitor and the needed configuration for this is out of the scope of this work.

Calculating the buffer size The parameters influence the buffer size required to collect the required samples specified by the operator. To foreshadow the connection between the parameters, the formula for the buffer size of node i labelled β_i is presented here:

$$N_{\gamma,\varepsilon} \leq r \cdot T \cdot \left((1-\delta_r)^{r\kappa} - (1-\delta_r)^{r\frac{\beta_i}{R_i}} \right) \cdot \sigma \tag{2.2}$$

In Equation (2.2), r represents the flow's rate, and R_i is the peak overall traffic rate observed at the node i. The red part represents the probability that the packet is not disclosed during the quiet period, and the blue part represents the probability that the packet does not suffer a late disclosure due to the buffer being too small.

The paper [4] also presents the formula for finding the ratio of the minimal buffer size $\frac{\beta}{R}$:

$$\min_{\delta_r \in (0, 1-e^{\frac{-1}{\kappa \cdot r_{min}}})} \left\{ \frac{\beta}{R} \right\} = \frac{\ln \left((1-\delta_r)^{r_{\min}\kappa} - \frac{N_{\gamma,\varepsilon}}{r_{\min} \cdot T \cdot \sigma} \right)}{\ln (1-\delta_r) \, r_{\min}} \tag{2.3}$$

Using the resulting ratio, the generally required buffer size can be obtained as follows:

$$\beta^* = R^* \cdot \min_{\delta_r \in (0, 1-e^{\frac{-1}{\kappa \cdot r_{min}}})} \left\{ \frac{\beta}{R} \right\} \tag{2.4}$$

Substituting R^* with the sampling node's peak observed rate R_i, the required buffer size β_i can be calculated. Hence, when changing from 2.5 Mpps (≈ 15 Gbit/s) to 25 Mpps (≈ 150 Gbit/s) of peak observable overall traffic, the buffer size also needs to be scaled, assuming the smallest flow's r_{min} remains identical. The RPS *paper* appendix can be consulted for more details.

2.5.1 Number of samples N

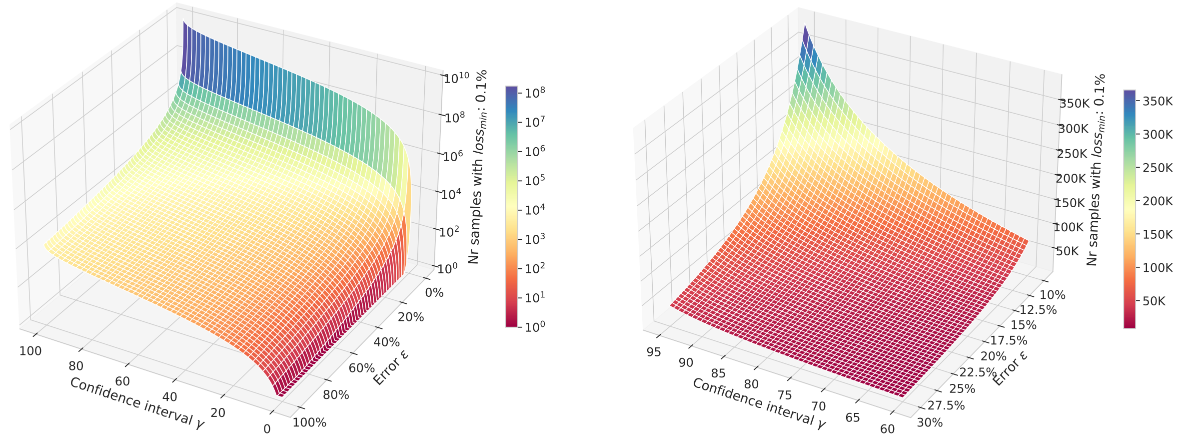
The first input parameter is the minimal number of samples N needed to show, whether an SLA is honoured. There are three distinct elements:

Accuracy The parameter of the accuracy is specified as a (γ, ε) tuple, where γ represents the confidence of the result whilst ε represents the error rate. In the words of the RPS *paper* authors: "γ is the probability that the relative estimation error is below ε" [4].

Loss Due to the sampling, the collection of a finite amount of observation, the lower bound for the loss must be set, such that enough samples are collected to form a statistically significant statement. A standard value for this $loss_{min}$ is 0.1%, as large ISPs such as *Cogent* guarantee a delivery rate of 99.9% [1,4].

Delay When delays are measured, these are referred to by their quantile χ. For example, the tail latency can be estimated using the 99-th quantile.

As the number of samples for small $loss_{min}$ ($\ll 50\%$) are larger than the required number of samples with $\chi > 50\%$, this parameter for the delay is omitted from now on and only the $loss_{min}$ is referenced [4]. With an $(95, 10)$ accuracy and a 0.1% minimal detectable loss, the minimal number of samples to collect is around $380'000$, requiring 9 MB of buffer space. Reducing the error level or increasing the confidence, the number of required samples drastically spikes up to 10^8 (Figure 2.3a), requiring large buffers, depending on the parameter choice, even infinite-sized buffers. Reducing the confidence or increasing the error, such as $(90, 10)$ or $(95, 12.5)$, leads to around $200'000$ required samples (Figure 2.3b) and a buffer size of 5.5 MB. Given that the minimal loss is determined in SLA, $loss_{min} = 0.1\%$ is seen as a constant and thus is not part of the parameter optimisation.



(a) Overall input space on a logarithmic scale, showing the drastic increase of the required number of samples with a low error rate.

(b) Zoom onto a realistic subset of target accuracies using a linear scale, showing the benefits of a reduction in the error level.

Figure 2.3: Number of required samples to achieve an (γ, ε)-accuracy under the loss of 0.1%. The x-axis represents the confidence, and the z-axis represents the error rate.

2.5.2 Collection time T

The collection time T determines the time interval for the collections of the N samples. The RPS *paper* suggests a value of 10 *min* as default. Hence, the monitor can estimate the loss or latency after T minutes with a (γ, ε) accuracy. At 2.5 Mpps and $\sigma = 1\%$, around 10 *MB* of buffer are required. Doubling the collection time reduces the buffer size by 2*x*, whilst further increasing the time eventually leads to a converging buffer size of 3 *MB* (Figure 2.4) for $\sigma = 1\%$ and $R^* = 2.5$ Mpps. The selection probability σ has a substantial influence on the buffer size for collection times below 20 *min*, as the buffer size for 6 *min* with $\sigma = 1\%$ is infinitely large, whilst, with $\sigma = 1.5\%$, only 12 *MB* are required.

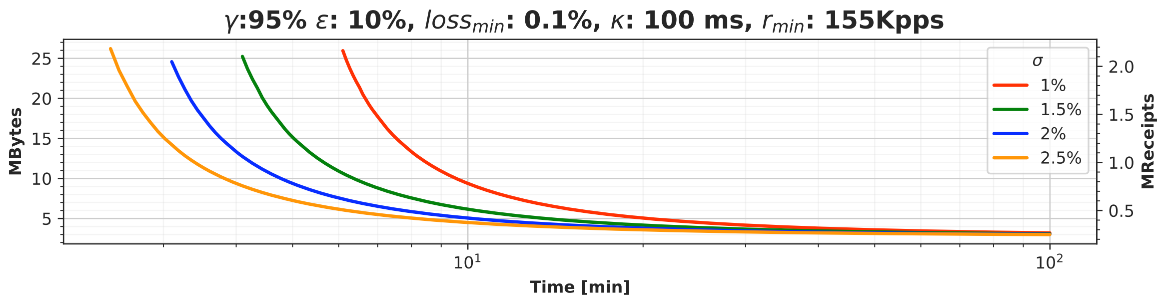


Figure 2.4: Buffer size based on the collection time T at $R^* = 2.5$ Mpps $\approx$ 15 Gbit/s

2.5.3 Quiet time κ

The quiet time also called quiet period κ, defines the minimal age of a receipt for a delayed disclosure to the direct disclosure. Combined with the jitter margin μ that aims to compensate for packet jitter and thus improve the consistent sampling performance, it is the key factor for ensuring resistance against prioritisation attacks. The argument, including the proof for the resistance, can be found in the RPS *paper*. The larger κ, the smaller the benefit of a prioritisation

attack, with the highest benefit if the ISP does not cheat. The *RPS paper* authors suggested 100 *ms* for κ for the current Internet usage. To determine a well-suited κ, the inequality $\kappa \gg d_b - d_g$ can be used where d_g is the latency of the best path and d_b the latency of the worst path of an ISP.

The buffer size increases linearly with the quiet time, roughly doubling from 8 MB at 100 *ms*, to 19 MB at 200 *ms*, reaching 100 *MB* at 1 sec with $\sigma = 1\%$ (Figure 2.5). Setting κ to 0 removes the requirement for a buffer and reduces the RPS Algorithm to a simple *trajectory sampler*.

Increasing σ results in noticeable smaller buffers for $\kappa \geq 200$ *ms*, around 2/3 of memory are needed at 1000 when σ is increased by half a percentage.

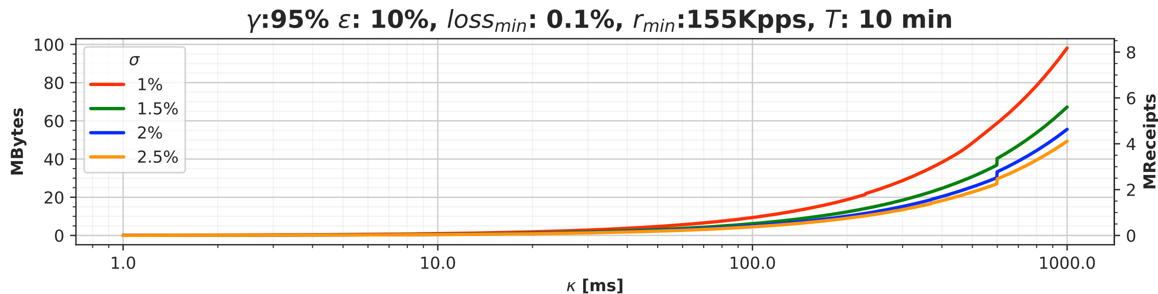


Figure 2.5: Buffer size based on the quiet time κ at $R^* = 2.5$ Mpps ≈ 15 Gbit/s

2.5.4 Minimal rate r_{min}

To guarantee the buffer is large enough for collecting the specified number of samples, a minimal rate r_{min} needs to be fixed. Every flow must always have a bandwidth larger than r_{min} to achieve the target sampling performance. If a flow has a high rate, there are too few direct disclosures chosen compared to the buffer capacity, and thus there are not enough slots to store all receipts. This results in fewer candidates for the delayed disclosure and, in the end, fewer direct disclosures. RPS does not allow for arbitrary small flows, as it requires a minimal flow rate of 40 to 90 Kpps, depending on the conditional selection probability σ, as smaller flow rates lead to the requirement of an infinite buffer size (Figure 2.6). Increasing r_{min} leads to smaller buffer sizes due to the higher traffic intensity, which holds more sample collection opportunities. However, the buffer size does not converge to 0, as some buffering is required to maintain the robustness against prioritisation attacks.

Again, the memory requirement can be reduced by over $2x$ by increasing σ from 1% to 1.5% at minimal rates below 160 Kpps.

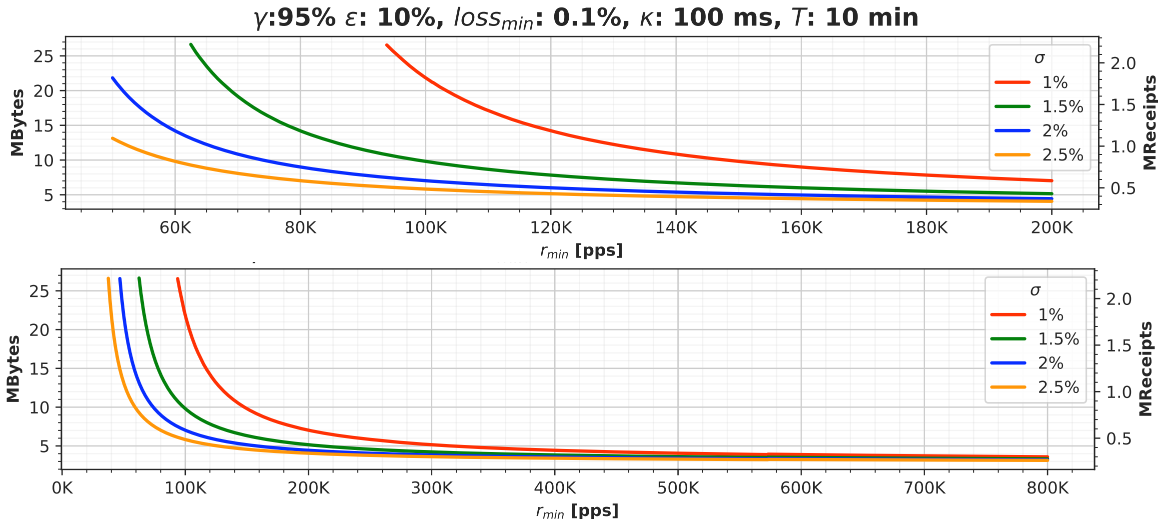


Figure 2.6: Buffer size based on the minimal rate r_{min} at $R^* = 2.5$ Mpps ≈ 15 Gbit/s

In the RPS *paper*, r_{min} is set to 155 Kpps, as this corresponds to a saturated OC-12 (Optical Carrier 12) link at an average packet size of 500 bytes. These OC links are commonly used in the backbones of ISPs [12]. Our analysis of CAIDA [13] traces from the year 2018 showed an average packet size of 800 bytes (Appendix A) and we will use $r_{min} = 200$ Kpps, resulting in 1.28 Gbit/s. This is equivalent to a fully utilized OC-24 or a semi-utilized OC-48 link [14].

2.5.5 Selection probability σ

The conditional selection probability σ determines the proportion of the delayed disclosure candidates that should be disclosed. With an infinite buffer, the σ determines how much overall traffic is sampled. Increasing σ leads to more collected samples and thus a smaller buffer size (Figure 2.6, Figure 2.4). In the RPS *paper*, it is suggested to use $\sigma = 1\%$, for the Internet as it is a rate technically supported by network devices, such as Cisco's IOS NetFlow [15].

Our analysis has shown the drastic impact of increasing σ for short collection times (≤ 6 *min*), quiet times larger than 200 *ms* and minimal rates below 160 Kpps. This could suggest that at least $\sigma = 1.5\%$ should be used to benefit from the reduced memory requirements. Whilst this is a good suggestion for sampling nodes observing a small traffic capacity, devices observing hundreds of gigabytes might not be able to support or sustain higher selection rates.

2.5.6 Disclosure probability δ

The direct disclosure probability δ determines how often a direct disclosure is generated and thus also defines the approximate frequency of clearing the flow's receipts. Due to the flow rate ranges, multiple δ can be used with thresholds. Together with the buffer size β, it is the only parameter that needs to be computed. To obtain δ, β has to be calculated first as the direct disclosure probability is chosen to minimise the late disclosures whilst achieving the required number of samples with minimal buffer size. Setting δ too high results in too few disclosures, as the proportion of the buffered receipts outside and inside the quiet period is too small, leading to too many samples falling into the quiet time. This occurs because the receipts are flushed faster than needed, and the following direct disclosure occurs earlier. If δ is set too low, receipts are

removed from the buffer due to the limited space, and thus, fewer samples are collected, possibly requiring more time to reach the targeted number of samples.

Regulating δ The *RPS paper* suggests using two to three values of δ to control the number of samples collected. In the base case, two values are used with a threshold, at which the δ is changed. With the main concern being collecting enough samples, δ is reduced if the flow rate exceeds a certain rate, reducing the number of collected samples. This is needed as the monitor only requires a fixed size of samples over a pre-defined time; collecting too many ($> 2x$) samples uses collection bandwidth without improvements for the target accuracy.

Finding δ Finding δ means defining an operational scheme, based on Equation (2.2) and a further inequality from the *RPS paper*. Given these equations, tuples r, δ that satisfy the equation's lower bounds can be found, and from this, an operational scheme can be made. As the *RPS paper* did not mention how exactly the thresholds were fixed, testing the parameters is needed to validate their efficiency.

2.5.7 Tofino™ parameters

The Tofino™ imposes strict memory size constraints as later introduced in Section 4.4.2, and during the evaluation, the reference parameters introduced in this chapter are used for compatibility reasons with other sampling nodes. Nevertheless, we present the maximal sampling capacity based on two cases: (1) The theoretically achievable performance based on the available buffer size scaling r_{min}, allowing up to 30 Gbit/s sampling capacity. (2) The performance under optimisation of the parameters, increasing the sampling capacity up to 48 Gbit/s.

Scaling r_{min} As previously introduced with r_{min}, the parameter is chosen based on the OC link capacity, resulting in a flow to link mapping. As links are upgraded with increasing capacity needs, changing the minimal rate can be a viable option for operators. With $\sigma = 1\%$, an increase to a minimal rate of 500 Kpps already leads to 4 Mpps (25 Gbit/s) of sampling capacity, doubling the capacity achievable at 155 Kpps (Figure 2.7). The returns diminish after 500 Kpps, converting towards a sampling capacity of 4.7 Mpps (30 Gbit/s) per pipeline. The ledges observable at around 575 Kpps and 950 Kpps are caused by the optimisation behaviour of the used Python optimiser.

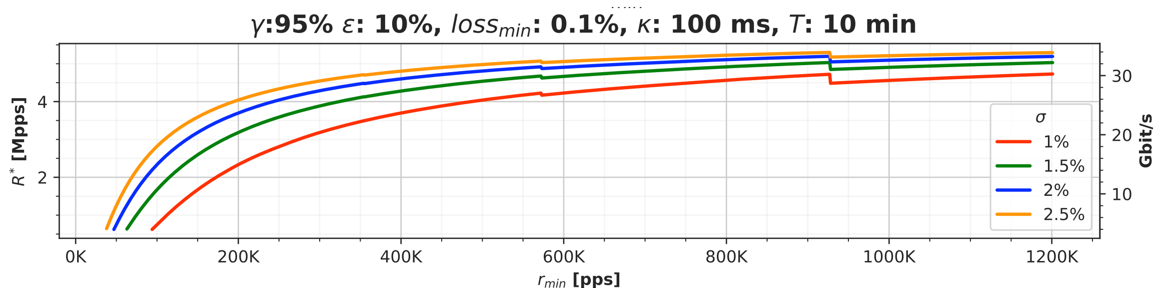


Figure 2.7: Sampling capacity with scaling r_{min} using the available 4 MB of receipt buffer available in a single Tofino™ pipeline. Rate based on an average packet size of 800 bytes.

Opimising parameters If parameter relaxation is possible, we propose a selected set that increases the sampling throughput by over 50%, showing already significant benefits for rates at 100 Kpps, allowing for a greater variety of flows to be sampled. These optimisations could be considered after increasing σ on both sampling nodes, or if the increase is impossible due to technical limitations. The first optimisation we propose is to reduce the accuracy from $(95\%, 10\%)$ to $(90\%, 15\%)$, by relying on the continuous arrival of data and the assumption of running the monitor over an extended period of hours. This is also the motivation behind increasing the collection time to 15 minutes. The monitor can continuously determine the metrics, thus stretching the collection time. Furthermore, we propose to reduce κ by one-fifth to 80 *ms* in favour of tracking smaller flows over a high prioritisation resistance. Using these parameters supports small flows, as even with 400 Kpps and $\sigma = 1\%$, 6.53 Mpps (41 Gbit/s) can be supported per pipeline. At 155 Kpps, 5.6 Mpps (35 Gbit/s) are supported, with larger σ even supporting rates below 50 Kpps at 4 Mpps (25 Gbit/s). Overall, the capacity converges towards 7.1 Mpps (55 Gbit/s).However, the difference in σ above 400 Kpps becomes insignificant.

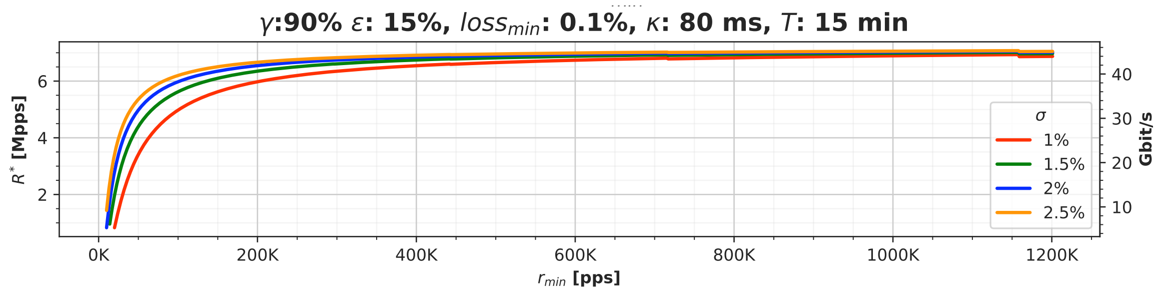


Figure 2.8: Sampling capacity with optimized parameters using the available 4 MB of receipt buffer available in a single Tofino™ pipeline. Rate based on an average packet size of 800 bytes.

2.6 Reference Implementation

For comparing all further implementations, a reference implementation was implemented to be as faithful as possible to Algorithm 2. As this is not about optimising the reference implementation, only a naive implementation was done.

2.6.1 Receipt

The receipt is the record stored per packet in the receipt buffer. As presented in the*RPS paper*'s appendix, they consist of 12 bytes: 6 bytes for the *flowid*, 2 bytes for the *timestamp* in the millisecond resolution and 4 bytes for the *digest* that identifies the packet.

Flow identifier The flow identifier *flowid* was initially chosen as the combination of the /24 prefix of the source and destination IP address. Although the size of 6 bytes was kept, experiments showed that the CAIDA traces could not satisfy the required per-flow minimal rates of $r_{min} \geq 155$ Kpps. Hence, it is assumed that defined flows satisfy r_{min}, even with this 6 bytes long *flowid*.

Digest The packet digest allows for the packet identification in combination with the *flowid* across multiple nodes; it is the packet's fingerprint. The *RPS paper*'s implementation computes the *digest* over the packet's first 48 non-mutable bytes. Ideally, this packet fingerprint is calculated

over the entire packet modulo the variable field, but this is a time- and computationally expensive operation in the network. The provided implementation deviates from both mentioned approaches. It uses the first 20 bytes of the IP header, the standard header without options, and the first 20 bytes of the TCP header if present. Removing the mutable fields of the IP header, such as the flags, fragment offset and checksum, the 20 bytes are reduced to 16 bytes. Routers can modify the IP flags and fragment offset, which changes the checksum and requires excluding these fields. This approach allows for 12 bytes (TCP) or 24 bytes (UDP) of application layer data to be incorporated into the digest, which improves the quality of the digest as later discussed in Section 4.3.1. To comply with the later implementations, instead of using a cryptographic MAC, as in the *RPS paper*, a CRC32 is used as a hash function to create the digest from the 48 bytes in this work due to the limitations in the programmable data planes. More details are discussed later (Section 4.3.1).

2.6.2 Buffer management

The buffer is implemented as a linked list, with each node containing a receipt and an indicator if the receipt is a disclosure. By inserting the direct disclosures in the buffer and setting the indicator, late disclosures can be detected without ambiguity compared to techniques introduced later. As these disclosures consume buffer space, the overall buffer size is dynamically adapted based on the number of disclosures in the buffer. Although a linked list is used, the memory requirements will solely mention the number of 12 bytes receipts and thus not the overhead of 8 bytes per node needed to keep track of the next node. The abstract data type of a linked list was chosen due to receipt deletions that can happen at arbitrary positions. A static buffer array would lead to empty spots and require additional tracking.

2.6.3 Expected performance

The *RPS paper* implemented RPS on an *x86* architecture using an Intel® Xeon E5-2680 processor and a 10 Gbit/s Network Interface Card, achieving a forwarding capacity of 14.14 Mpps with an average extra processing time per packet of 1.35 *us* [4]. Our implementation reaches around 1 Mpps, however, the performance is not directly comparable for three reasons: (1) as introduced, our implementation is not optimised and its only optimization is the compilation with `-O3`, (2) our implementation relies on reading PCAP [16] files instead of reading the raw packet data from a highly optimised Data Plane Development Kit (DPDK) [17] buffers and, (3) the CRC32 is purely computed in software [18], not leveraging hardware support, whilst the *RPS paper* uses specialised AES-NI [19] instructions available in the used processor.

Chapter 3

Programmable Data Planes

Programmable data planes removed long, inflexible and costly processes to create custom Application-Specific Integrated Circuits (*ASIC*s) by providing a Protocol Independent Switching Architecture (*PISA*) that supports a programming language called P4 to modify headers. Devices supporting P4 allow software-defined data planes and operators to test and implement new protocols within hours. Additionally, these devices can process packets within hundreds of nanoseconds while maintaining line rate. However, P4 has limitations such as not having freely addressable memory and no unbounded loops as one is used to from high-level programming languages such as Python.

3.1 PISA

As network switches are based on highly specialised *ASIC*s, architectures may vastly differ between vendors and products. To allow for efficient code deployment, a shared abstract model of the hardware is required. Models are tailored to specific needs, and thus, a wide range exists. One is the Protocol Independent Switching Architecture (*PISA*), which is commonly used. It consists of a programmable parser, which is followed by a match-action pipeline and terminates with the programmable deparser [20].

Parser and Deparser To process an incoming packet, programmers need to know the structure of the information contained in it. Hence, *PISA* contains a parser which extracts headers and separates them from the payload and a deparser which combines the dissected packet before sending it. As both the parser and deparser are programmable, the dissection of an incoming packet and the re-combination are customisable in software. In the example use case of processing TCP packets, one would extract the Ethernet, IP and TCP headers.

Match-Action Pipeline The match-action pipeline is the core of the packet processing, where the operations on the packet headers are defined. Program-defined parts of the headers are passed to the logic units called match-action units, containing match-action tables (*MAT*), which result in the modification of the headers, by matching data to functions that execute instructions to modify metadata or headers. These Look Up Tables (*LUT*) allow for efficient and fast packet processing. Although a wide range of operations can be represented, the pipeline abstraction requires programs to be representable as a Directed Acyclic Graph (*DAG*). In many models, the Match-Action pipeline is split into an *Ingress* pipeline, which treats packets after receiving them

but before the port assignment decision (and respective queuing) and an *Egress* pipeline, which treats the packets after the port assignment and queuing [21].

Control plane and Data plane Until now, we have seen components that belong to the data plane. More precisely, the dataplane is in charge of forwarding the packets and applying the defined operations, such as reducing the IP TTL. In contrast, the control plane is in charge of defining how the traffic is forwarded. As explained in the example of a link-layer switch, the control plane defines which destination address matches which switch port, whilst the data plane is responsible for resolving the destination port and sending the packet over this port.

3.2 The P4 Language

So far, we have specified an abstract architecture of the programmable switches, and we now move on to the programming language used to orchestrate and populate the parsers and match-action pipeline. Programming Protocol-independent Packet Processors (*P4*) [5] is a commonly used high-level language for programming *PISA* data planes [20]. It is a domain specific language consisting of less than 40 keywords [22], and provides a concise abstraction of the programmable switching architecture independent of the underlying hardware. P4 Code is compiled into hardware-specific representations that are understood by the target switching device. Compared to a general-purpose programming language such as C, P4 does not offer unbounded loops due to the acyclic representation and strict time budgets. Furthermore, no allocatable memory and pointers exist as the state is generally only kept as packet metadata, carried from the parser until it reaches the deparser. As rate limiting or packet counting requires more than a per-packet state, P4 offers *externs*, which can hold limited information. Information such as routing tables written by the control plane is stored in the switch's Ternary Content-Addressable Memory (TCAM) or Static Random-Access Memory (SRAM).

P4 targets To run P4 code on a device, it must support the *PISA* and ship a compiler that transforms a program into a hardware-specific representation. As the hardware design choices influence the P4 programs, target manufacturers provide the abstractions as P4 code in so-called *models*. Two notable targets used in this work are the *BMv2* and the Intel® Tofino™.

BMv2 The Behavioral Model version 2 (BMv2) [23] is the reference software switch for P4, and it is designed for testing purposes. Written in C++ for quick modifications and debugging, it runs on many platforms and does not require specialised hardware. However, it is not designed for production environments and can only reach up to 1 Gbit/s in throughput with all logging facilities disabled. The hardware abstraction is the *Simple Switch* built on top of the *v1model*.

Tofino Intel's Tofino Native Architecture (TNA), which is available on their Tofino switching ASICs [24], is a hardware solution used in industry applications. Released in 2016 (Tofino™) and revised in a second version in 2018 (Tofino™ 2), the latest revision supports up to 32 ports at 400 Gbit/s, leading to a throughput of up to 12.8 Tbit/s and a frame processing rate of 6 Bpps. Programmers can populate 20 stages in the ingress and egress pipeline, and the ASIC has a packet buffer of 64 MB [25]. Installed in a switch, an added latency below 400 ns for a 64 port switch with a throughput of 6.4 Tb/s [26] can be expected. It provides a large set of accessible intrinsic metadata and external functions such as customisable CRC checksum functions [20].

Edgecore Networks WEDGE100BF-32Q This work uses a Edgecore Networks WEDGE100BF-32Q as a hardware testing platform. It is a 32 QSFP port programmable network switch with a port speed of up to 100 Gbit/s built around the Intel® Tofino™ ASIC with 2 individually programmable pipelines each with 12 stages (Figure 3.1). It offers a throughput of up to 3.2 Tbit/s, 22 MB of packet memory and a peak frame processing rate of 4.8 Bpps [27]. The ASIC is collocated with a *Bare Metal Controller (BMC)* consisting of an Intel Pentium D-1517 4 core processor clocked at 1.6 GHz with 8 GB of DDR4 RAM and 128 GB M.2 SSD. The Tofino™ is integrated into the processor's root complex over a PCIe Gen2 x4 bus. Additionally, two Intel X552 10 Gbit/s network interfaces connect the BMC to the ASIC.

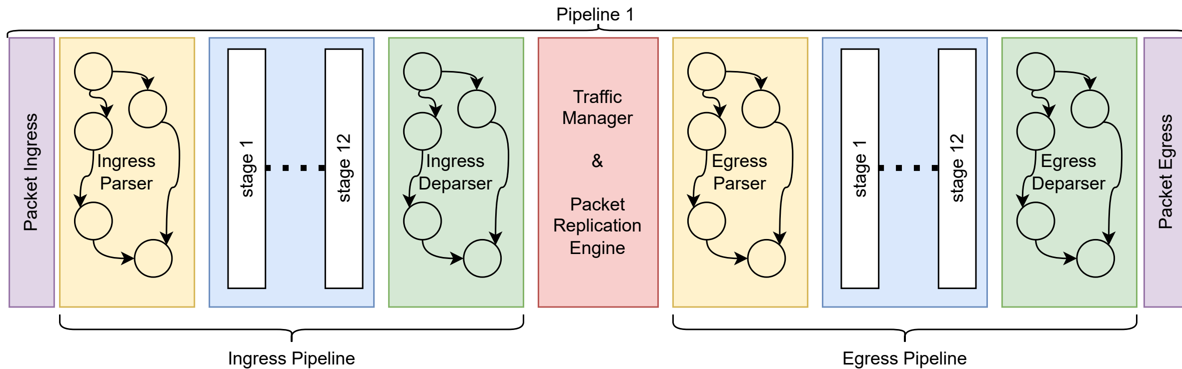


Figure 3.1: Structure of the TNA. In addition to the introduced PISA, the Ingress pipeline has a deparser and the Egress pipeline has a parser. Pipeline 2 is identical to pipeline 1.

3.2.1 P4 Features

As the *PISA* and *P4* enforce constraints on the available programming features and thus limit the expressible algorithms, two key differences to a general-purpose architecture relevant to this work are highlighted. These are the storage and traffic manipulation primitives.

3.2.2 Storage

As mentioned before, P4 does not directly offer memory to store data. However, there are different storage primitives available, such as *tables*, *registers*, *counters* and *meters*. The tables are supported natively by P4 whilst all other primitives are implemented as `externs`. Aside from the efficient tables, the register is the most powerful storage primitive (Table 3.1).

Feature	Type	Data Plane	Control Plane
Table	native	read	read & write
Register	extern	read & write	read & write
Counter	extern	write[1]	read & write[2]
Meter	extern	read [3]	read[4] & write[5]

Table 3.1: Overview of the different storage primitives available in P4.

[1]The value can be incremented using `count()`.
[2]Often, only clearing is allowed as a write operation.

Table The tables build the foundation of P4 and are the core abstract datatype for the control plane to dynamically change the effects of the data plane. This storage primitive is the implementation of the previously presented MATs. The lookup key can consist of multiple fields/variables and resolves to a pre-defined set of actions. As shown in Listing 3.1, each key has a distinct match type: The source address is matched using the longest prefix match (*lpm*), allowing to match to the most specific IP address present in the table, the header length field is matched using a *ternary* match, in which a bitmask can be used to define bits to be ignored. The IP version is matched using an *exact* match type, meaning that the value to be looked up has to agree exactly. Finally, the TTL is matched within a defined range as the *range* match type is used.

Tables can only be modified by the control plane. The data plane is optimised for finding table entries and executing the associated actions.

```
table example_table {
    key = {
        hdr.ipv4.srcAddr:   lpm;
        hdr.ipv4.ihl:       ternary;
        hdr.ipv4.version:   exact;
        hdr.ipv4.ttl:       range;
    }
    actions = {
        set_destination_port;
        drop;
    }
    size = 256;
    const default_action = drop;
}
```

Listing 3.1: Example of a P4 table.

Register Registers are an `extern` functionality that offers limited storage that can be accessed from the data plane as well as from the control plane. All metadata and packet data collected during a packet's live time in the data plane is lost after the packet processing, due to the data plane being stateless. However, some applications need persistent states, such as tracking the time between two requests. In these cases, the data plane can determine an index in a register array and write a timestamp to this register field. When a new request comes in, the index can be again determined, and the previously stored value can be read out and processed.

Counter Counters are a further `extern` specifically designed to count packets or bytes for the control plane in the data plane. They are also allocated in an array and the data plane increments the counter at a given index by calling the count function. Furthermore, counters can also be directly associated with tables, using the table's index as a counter index. This can be used to count how many times an action was executed.

Meter Meters allow the measurement of rates using a standardized tracking mechanism [28], resulting in one of the three colours: red, yellow, and green. Depending on the arrival rate and the operator's configuration, a packet is coloured, and the colour is made available to the data plane.

[3]The meter state is returned, and the meter is automatically updated.

[4]Cannot read the bucket states; only the parameters are available.

[5]Only parameters can be set.

This feature is intended for traffic shaping, such as rate regulation of packet streams. Similarly, to counters, meters are indexed or associated to a table as *direct meters*.

3.2.3 Traffic manipulation

In addition to processing packets passing through P4 devices, multiple options to modify the traffic exist, such as *recirculating*, *mirroring* or sending packets or metadata to the controller.

Recirculation If complex operations on a packet need to be executed that cannot fit into the time or resource budget, such as encrypting packets in the data plane [29], the packet needs to be sent from the egress to the ingress again for an additional of processing, retaining all changes made to the packet. Furthermore, operations such as multicast might require the recirculation of a packet to be sent to different destinations. Hence, packet recirculation is an important feature.

Mirroring Packet mirroring slightly differs from recirculation, as the packet is copied as it was received, and the duplicate is then processed individually, compared to the recirculation, which keeps the changes to the packet. Mirroring allows packets to be duplicated and sent to a different location for additional processing.

Digests The digest feature in P4 is designed to share small amounts of data with the controller. One use for this feature is the MAC learning during the Address Resolution Protocol (ARP) [30], where the switch learns of the neighbours and needs the control plane to populate the tables with the learned data to allow packet forwarding.

Chapter 4

Implementation

This chapter introduces the implementation decisions and challenges faced while adapting the RPS Algorithm to P4 and the Tofino Native Architecture. Due to the restrictive memory primitives, we adapted the algorithm to only remove the oldest receipt from the buffer instead of receipts at arbitrary positions.

After defining the design goals, various implementations are presented. Firstly, sampling with a close relationship to the controller is introduced, followed by implementations that use the data plane memory to buffer receipts. Lastly, the adaptations needed for the Tofino™ are introduced.

4.1 Implementation Goals

We must fix the implementation goals to design a P4 implementation of the RPS Algorithm. This is especially important since the RPS is not yet standardised and only exists as an academic publication. Hence, we first define our goals and then mention the non-goals.

4.1.1 Goals

To provide a real-world implementation of RPS on network switches, we want it to be compatible with the reference from the *RPS paper* as well, and it should be runnable on an Intel® Tofino™ ASIC.

Accuracy The first and most important goal is the compatibility of the P4 implementation to the RPS Algorithm provided in the *RPS paper*, and thus providing an identical accuracy. Concretely, the designed algorithm needs to be able to run in a network of RPS nodes independently of the implementation. Hence, the designed algorithm shall produce comparable disclosures as the RPS Algorithm under the identical base configuration.

Sampling in the dataplane To create an application that scales with traffic rates in the order of hundreds of gigabits, we want to minimise the bandwidth usage between the data plane and the control plane. Hence, we set the goal to push the complexity of the RPS Algorithm into the data plane as much as possible.

No additional hardware To incentivize ISPs to deploy RPS, they cannot be expected to spend thousands of dollars to buy new hardware just to run the proposed implementation. Hence, the

design should avoid using additional hardware that is not packed with a P4-enabled switching ASIC.

Deployable implementation In addition to the goal of not requiring additional hardware, we also want our proposed design to be deployable on a Tofino™ ASIC. This does not mean that the evaluated implementations are production-ready, but that they are at least runnable on the mentioned target architecture instead of a generic P4 implementation for BMv2. This goal is mainly based on our challenges while migrating our reference implementations from the Simple Switch target to the Tofino Native architecture. The provided implementation shall serve as a feasible base implementation for experienced P4 developers.

4.1.2 Non-Goals

As RPS is a broad topic, we want to focus on providing a P4 implementation and not on the algorithm's behaviour and accuracy. This also includes omitting the entire receipt aggregation framework needed to collect the receipts from various ISPs.

Analyzing RPS Although an analysis of the RPS Algorithm outside of the initial *RPS paper* publication would be interesting, we explicitly avoid this. This is due to the goal of providing an implementation compatible with the RPS Algorithm in the *RPS paper*. Furthermore, to our knowledge, the RPS Algorithm is not yet used in production and is also not standardised; thus, not all edge cases are covered.

Full RPS infrastructure RPS is more than just the RPS Algorithm: there is also the aspect of collecting the initial metadata as well as aggregating and evaluating all disclosed receipts in the monitor as seen in Section 2.3. To focus on the actual implementation, we assume that a framework is in place to collect and evaluate the created receipts and ensure that the loss of receipts can be accounted for correctly. Additionally, we assume the monitor to be capable of determine the required traffic metrics based on the collected receipts.

4.2 RPS on the Controller

Although we have set the design goal to push as much of the RPS Algorithm into the data plane, a simple option is to send the first 48 non-mutable bytes, which are used to create the digest, to the controller. This approach has the advantage that the buffering behaviour is identical to the reference implementation from the *RPS paper*. A second approach is the creation of the receipt in the data plane and then forwarding it to the controller, reducing the number of bytes per incoming packet that must be sent to the controller from 50 to 12.

4.2.1 Header mirroring

A simple implementation is to clone the incoming headers and then forward them to the controller over the network interfaces connecting the ASIC with the host system. The advantage of this approach is the simplicity, whilst the disadvantages are the limited performance and the violation of the design goal of running the algorithm in the data plane.

Implementation Switch Every incoming packet is cloned during the ingress processing and then forwarded to the host. To save some bandwidth, the Ethernet header, as well as the first non-mutable 48 bytes of the following headers, are sent to the control plane as shown in Figure 4.1. The resulting Ethernet II frame has a size of 68 bytes due to the 14 bytes long MAC header, the 2 bytes for the timestamp of the packet arrival and the first 48 non-mutable bytes. The last 4 bytes are used for the Frame Check Sequence (FCS). To save bandwidth, the source MAC field is used to transmit the 6 bytes long *flowid*. As two NICs exist in the host system of the Wedge 100BF-32Q, load balancing is performed.

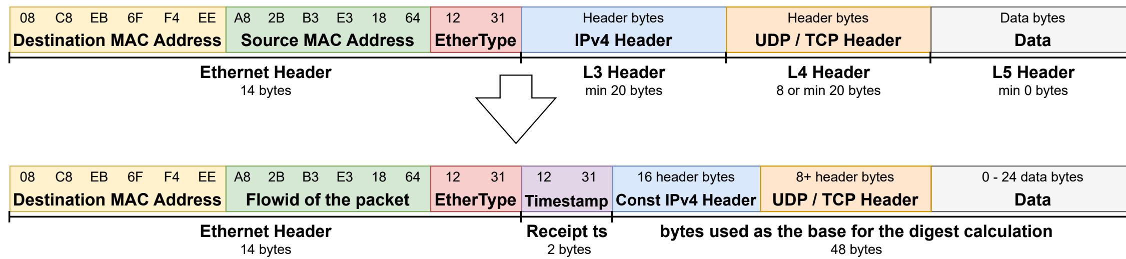


Figure 4.1: Overview of the header fields to extract to build the packet sent to the control plane.

Implementation Host The packets are received and processed on the host side with the reference implementation of the RPS Algorithm. The host listens to the interfaces connected to the Tofino™ ASIC and extracts the incoming packets. After constructing the 12 bytes long receipt is inserted into the buffer based on the rules set in the reference RPS Algorithm.

Performance estimation Based on the two available 10 Gbit/s NIC on the Wedge 100BF-32Q and assuming full utilization of these, at a peak rate of 36.764 Mpps can be handled. The number of processable packets is based on the total interface rate of 20 Gbit/s and the packet size of 68 bytes sent to the host for processing. Assuming an average packet size of 800 bytes, around 235 Gbit/s of traffic could be handled. However, it is highly unrealistic to achieve this due to the 100% load of the two NIC. Utilizing the results of the *RPS paper*, with an achieved performance of 14.14 Mpps per 10 Gbit/s NIC, a sampling performance of 28.28 Mpps or equivalently 180 Gbit/s is realistic.

Tradeoffs Whilst this approach is not bound by the memory limitations of the Tofino™ itself, the main limitation is the bandwidth between the Tofino™ ASIC and the host system. If we discard the goal of no additional hardware, the switch's multiple 100 Gbit/s ports can send the base packets to a dedicated server with corresponding NICs. However, every cloned packet reduces the Tofino™'s peak rate of 4.8 Bpps [25], and cloning every packet reduces the peak forwarding capacity by a factor of two. Furthermore, are all packets of size near the minimal Ethernet II size, which might complicate the high utilization of high-speed NICs optimized for transmitting large frame sizes such as jumbo frames.

4.2.2 Digest mode

Whilst the previous attempt sent 64 bytes to the controller that is reduced to a 12 bytes large receipt, this approach reduces the overhead by creating the receipt in the data plane. It uses the P4 digest feature to transmit the receipts in batches to the host.

Implementation Switch In contrast to the mirroring over the NIC connected to the host system, this implementation calculates the *flowid*, *digest* and *timestamp* directly in the data plane and only forwards the 12 bytes large *receipt* to the host CPU. On the Tofino™ ASIC, the *Learning Filter* feature is used, which allows the creating of 48 bytes large messages called *Learning Quantas (LQ)*. Up to 2048 unique LQ are stored in a buffer either periodically or upon being full, transferred per Direct Memory Access (DMA) to the host system. This effectively reduces the transmitted bytes per *receipt* from 68 to 12 when compared against the mirroring approach, excluding possible annotations used internally for the DMA transfer. As the Tofino™ cannot compute SHA256 hashes or CBC MACs, the CRC32 checksum is used for the *digest* creation, which will be discussed in Section 4.3.1 in detail.

Implementation Host As the *receipts* are transferred over DMA, the host can directly insert the *receipts* into the buffer and thus run the RPS Algorithm without creating a new *receipt* from the raw bytes. This reduces the computational effort and the number of bytes to be processed.

Performance estimation Based on the PCIe Gen 2 x4 connection between the Tofino™ ASIC and the host system of the Wedge 100BF-32Q, a peak rate of 166.667 Mpps can be handled under the assumption of a fully utilized connection. The number of processable packets is based on the PCIe 2 x4 link that provides 5 GT/s with an 8b10b encoding, leading to a usable bandwidth of 16 Gbit/s and the *receipt* size bytes that is sent to the host for processing. Assuming an average packet size of 800 bytes, around 1 Tbit/s of traffic could be theoretically handled. However, such a high utilization of the PCIe interface seems unreasonable, as the entire control of the Tofino™ ASIC is performed over this interface. Furthermore, the system with 8 GB of RAM and a 4 core Intel Pentium D-1517 clocked at 1.6 GHz base-frequency might become unstable or is unable to handle 4 GB/s of *receipt* data. Assuming a 85% utilization, 141 Mpps could be processed, resulting in a traffic rate of around 902 Gbit/s at an average packet size of 800 bytes.

Tradeoffs Similar to the approach using the mirroring of the headers, this approach is not bounded by the limited Tofino™ memory. However, the complexity of the algorithm is pushed into the control plane, which contradicts the goal of running the RPS Algorithm as much as possible in the data plane. Furthermore, the high utilization of the PCIe interface might negatively impact other ISP services running on the switch that need to communicate with the Tofino™.

4.2.3 Interim conclusion about RPS on the controller

So far, two designs have been presented, which both avoid the challenges caused by the limited storage capacity of the Tofino™ ASIC by pushing the complexity of the storage to the host system. Whilst these designs should allow for traffic rates of over 100 Gbit/s, they do not follow the goal of pushing as much complexity as possible into the data plane, as they occur a bandwidth overhead. Nevertheless, mirroring reduces the bandwidth between the data plane and the sampler by 12.5x, assuming an average packet size of 800 bytes. The packet rate remains constant. Ignoring DMA overheads, the reduction is 66.6x for the digest implementation.

4.3 RPS on the Simple Switch

To bring the RPS Algorithm onto the Wedge 100BF-32Q, we use the intermediate step to implement a version for the *Simple Switch* target that runs with the *BMv2*. This intermediate step allows us to

resolve the challenges of the loop within the RPS Algorithm and the lack of directly addressable memory. Furthermore, we present a workaround for the hash functions, as cryptographical hash functions are generally unavailable in P4.

4.3.1 From SHA256 to CRC32

Whilst the authors of the *RPS paper* used a Message Authentication Code that is based on the AES block cypher as they had a CPU with hardware support for this operations [4], it is clear that this is not possible in a generic P4 setting. Hence, we evaluate the *Cyclic Redundancy Check (CRC32)* checksum algorithm as a substitute and show the faced tradeoffs. As the hash function selects candidates to disclose, the *CRC32* must produce a closely related distribution of hashes. We find that the *CRC32* produces a similar distribution and thus can be used as a replacement, whilst the cryptographical properties of a *CMAC* or *HMAC* are lost.

CRC32 The CRC32 is a CRC that produces a 32 bit-wide result by performing a polynomial division on the input data. The application can define the 33 bit large polynomial; nevertheless, some polynomials are standardized. One example of this standardization is the ISO 3309 CRC-32, used as the Frame check sequence in the *Ethernet* protocol standardized as 802.3. As every NIC uses this CRC, specialized hardware is available to perform these calculations efficiently.

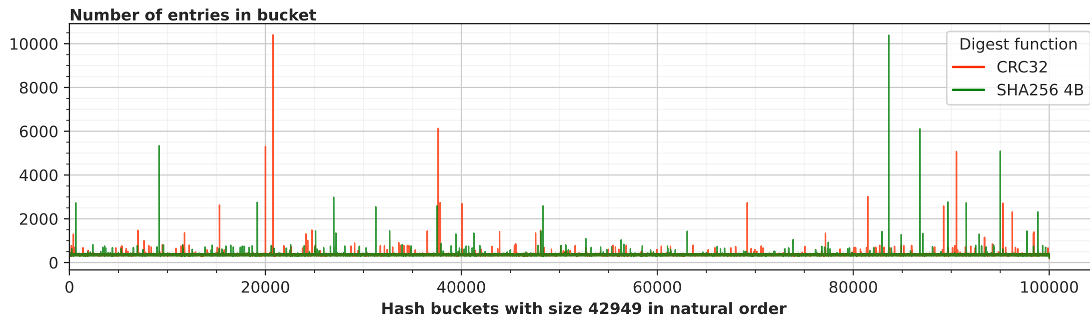
Difference between MAC and CRC Although both, CRC and MAC aim to detect modifications of the provided data, they fundamentally differ from a security perspective. CRC is designed to ensure data integrity during transmission without the assumption of an active attacker, whilst MAC aim to prevent the modification by an attacker by providing integrity and authentication. Although *CRC32* is technically a checksum, we refer to it as a hash function or hash algorithm in this work.

Impact on the RPS Algorithm The main motivation of the *RPS paper* to use a MAC was the availability of hardware acceleration and the randomization properties of these cryptographical building blocks. When using the *CRC32* algorithm, the cryptographical properties are lost. However, we leave the analysis of this impact to future work. Nevertheless, we can conclude that the reversing of the *CRC32* is possible [31] and thus allows for attacks on the RPS Algorithm. Neglecting the security aspect, we only require the 4 bytes of the MAC to provide a similar distribution of the digest as *CRC32* would produce.

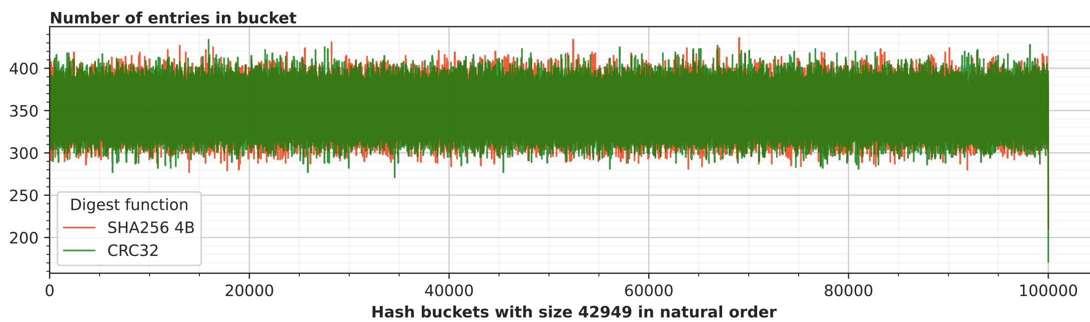
Uniformity of the hashes based on real data To understand if the different methods of digest generation result in a uniform distribution, we analyze a CAIDA trace by generating the digests and plotting their distribution. However, the distribution is not uniform; *SHA256* and *CRC32* produce similar anomalies.

With 32 bit large digests, 4.3 billion different hashes are possible. To understand the uniform distribution of the hashes, the number of entries in a total of 100′001 bins is compared to the number of entries of the given bucket for a trace with 35 million packets. As a data source, the *equinix-nyc.dirA.20190117-130000.UTC.anon* CAIDA [13] trace is used as it reflects a realistic scenario of packet headers encountered by a switch. If the hash functions are perfectly uniform, every bucket is expected to have 350 entries and thus a constant horizontal line for both hash functions. However, the distribution contains multiple spikes for both *SHA256* and *CRC32*. These spikes occur similarly for both hash functions, so the underlying data must be considered

(Figure 4.2b). The basis for the hashes are the first 48 non-mutable bytes from the *IP* header upwards. However, the CAIDA traces only provide the Internet and Transport Layer headers, which leads in the case of UDP to 20 bytes of zeroed data. Hence, the application layer's data would be considered, leading to a better distribution. To sum it up, the large collision spikes under both hash functions are an artefact of repeating data rather than issues with the function itself. The effects are significantly reduced if application layer traffic is used, with the number of entries falling within a range of ±75 entries, compared to the previous spikes in the thousand of entires (Figure 4.2b).



(a) Without application layer data



(b) With application layer data

Figure 4.2: Distribution of the 4 bytes long digests from the CAIDA [13] trace. The x-axis indicates the bucket number, whilst the y-axis indicates the number of entries per bucket. The colours indicate the used hash function. As spikes in the distribution exist, the distribution is not perfectly uniform.

Distribution similarity Whilst both hash functions produce similar anomalies, it remains to check if both produce a comparable distribution of hashes. As the RPS Algorithm relies on the hash distribution for sampling, *CRC32* should match as closely as possible the *SHA256*. When evaluating the distributions based on CAIDA data, it is found that *CRC32*'s distribution matches closely to *SHA256*'s distribution. The empirical cumulative distribution function (ECDF) is plotted to compare the distributions of the two hash functions in Figure 4.3. As both curves overlap and cannot be distinguished by eye, the distribution can be labelled similarly.

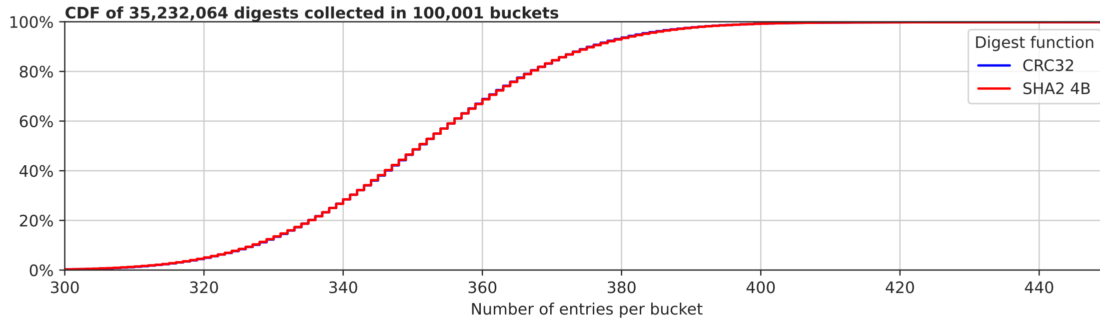


Figure 4.3: ECDF of *SHA256* and *CRC32* based on the CAIDA trace *equinix-nyc.dirA.20190117-130000.UTC.anon* containing 35 million packets, hashed into 100′001 buckets.

4.3.2 Buffer access in P4

A core component of the RPS Algorithm is the deletion of all receipts of a given flow as soon as a candidate is picked as a direct disclosure. As P4 does not allow for dynamic loops and does not offer freely addressable memory, a different approach than a generic linked list is required. Addressing this challenge, we adapt the RPS Algorithm by 'inverting' it. Instead of removing receipts upon determining a direct disclosure, we remove the receipts and determine delayed disclosures upon the arrival of new receipts. This approach allows for a high degree of decoupling between the data plane and the control plane whilst maintaining the sampling properties.

Storing receipts in the data plane In order to store receipts in the data plane, there is only a single option: registers. The register `extern` is the only `extern` that allows writes and reads directly from the data plane as introduced in Section 3.2. Hence, a register array stores the receipts as a whole or as individual fields. As there are no viable alternatives, the register array in the data plane is from now on referenced as *buffer* and the register index referred to as *buffer index*.

Removing the loop dependency To migrate the RPS Algorithm from general purpose language such as C or C++ to P4, which does not offer unbounded loops, an alternative approach to removing all packets from a flow upon a direct disclosure has to be found. But why does P4 not offer unbounded loops? As introduced in Section 3.2, resource and time constraints make unbounded loops nearly impossible, especially when processing at line rate. Furthermore, their number of stackable headers is often bounded when dealing with network packet headers, or not all header fields need to be parsed. Other frameworks, such as *Flare* [32] that also offer payload processing, do not have these restrictions but suffer from other limitations.

Looking at the RPS Algorithm, we find that the number of receipts to remove from the buffer can range from 0 to thousands of receipts, given that δ is usually in the range of 10^{-5} which makes a static unrolling impossible on the *PISA*. Hence, a P4 implementation of RPS should change the structure of the RPS Algorithm such that the dependency in the data plane from one incoming packet to looking directly at n packets is changed into a 1:1 mapping: one incoming packet triggers the modification of a single[1] buffer entry.

[1]This constraint could be weakened by allowing a 1:x mapping with $x \in [1, \approx 10]$, however, the Tofino™ architecture introduced later will impose strict bounds on x.

4.3.3 Naive implementation

To understand the limitations in P4, we start with a naive implementation closely matching the reference RPS Algorithm. This approach solves the challenge of implementing the RPS Algorithm without loops with a list of actions to be taken upon receiving a new receipt.

To remove the need for looping over the buffer in the data plane, the naive approach pushes this complexity into the control plane such that the data plane can directly determine the buffer index at which a new receipt is to be stored. Abstracted, the controller provides instructions from the control plane as a list of the following entries and actions to be taken on the data residing at this entry. The data plane becomes a queue consumer, and the controller manages the queue as a producer by adding entries - if needed at arbitrary positions.

To dive deeper, we first start by looking at the controller's duties and the look at the data plane's actions. The entire control flow is shown in Figure 4.4.

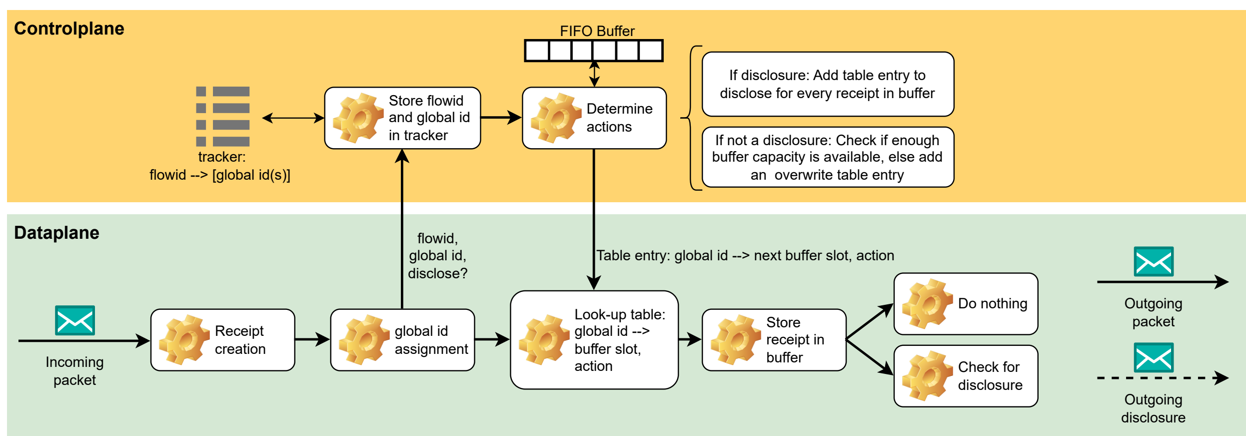


Figure 4.4: Control flow of a naive implementation of the RPS Algorithm in P4. The control plane manages the buffer, and the data plane adds and removes entries from the buffer. Removed receipt entries can be checked for delayed disclosure or dropped. A global identifier in the form of an incrementing counter is used such that the data plane can find a corresponding action.

The controller maintains the information of the incoming receipt order as well as the mapping of *flowids* to the buffer indices. It receives the *flowid*, a global identifier for every incoming packet, and the information if this referenced packet's receipt is a direct disclosure. Based on this information, the controller determines which buffer index will be used next and what action will be taken with the data residing at this entry. If a direct disclosure triggers the eviction of all receipts of the flow, the controller will add the buffer indices at the front of the task queue with the additional information that every receipt, before it is overwritten, should be checked for a delayed disclosure. Figuratively speaking, instead of unallocated memory, memory no longer needed is re-used. Every newly incoming packet removes an 'old' entry, and its content might be checked for delayed disclosure.

Before we move to the data plane, the yet unexplained global identifier has to be discussed. As there is no abstract data type such as a queue in P4, a counter, which is incremented upon creating a receipt, is used as the data plane's pointer to the next task to work on in the queue filled by the controller.

The data plane is straightforward. A receipt for an incoming packet is also generated, and the global identifier is retrieved. The information about the current receipt is passed to the controller,

and the buffer index is retrieved by using the global identifier as a look-up key in the task table, which serves as the controller's queue. The entry is overwritten based on the registered action, or a check for a delayed disclosure is performed.

Shortcomings of the Naive approach Although this simple approach might seem well suited, there are a few drawbacks to it, with the main one being the high reliability of the controller. For this approach to work, the following factors need to be taken into consideration:

Information flow The controller runs the reference RPS Algorithm, except it stores the buffer index, which is in *C*-terms just a pointer to the allocated receipt in the data plane, instead of the *timestamps* and *digests*. Therefore, the global identifier, the *flowid* and the disclosure status, must be sent to the controller for every generated receipt. With a x bytes wide *flowid*, $\frac{x}{x+6}$ of a receipt are sent to the controller every time. With a 6 bytes wide *flowid*, this ratio is 50%, begging the question of why not just sending the entire receipt to the controller as introduced in Section 4.2.1.

Low delay As the controller is in charge of populating the task queue of the data plane, the unique identifier should ideally not move during the controller's processing, as the controller might already replace the next task that should be performed. An example of this is if the case of a full buffer and the incoming receipt r_x causing a direct disclosure for *flowid* f_x: We would expect the buffer to have the capacity for the following receipts r_y as all receipts of f_x are on the list to be removed and this 'free'. However, if the delay between the r_x and r_y is smaller than the time it takes for r_x's information to be sent to the controller, being processed and (re)writing the table entries to the data plane, the data plane's global identifier will have moved on. It drops the oldest receipts in the buffer based on the table entries. Even if this is considered and some drops due to the delay are allowed, the controller still needs to predict the next global identifier, which is meaningful to replace.

Table updates As a direct disclosure alters the task queue, many table entries might be modified, as the queue is implemented as a table with the global identifier as the key. Inserting n receipts to be removed means either reducing the global identifier x in the data plane by n and inserting the receipt tasks at the indices $idx \in [x - n, x]$ in the table without touching the later tasks or by shifting all existing tasks in the table by n. As, in reality, updates are not directly visible; the controller has to anticipate how far the global identifier has moved and needs to perform the needed compensation. Furthermore, research indicates that the Tofino™ peaks at $100K$ table modifications per second [33], setting a hard limit on the scalability of this approach.

Looking at these three points and referencing them with our design goals, we find that we violated the goal of pushing as much complexity as possible into the data plane due to the heavy reliance on the control plane to update the task list. Furthermore, we found that the naive implementation is sensitive to delays and requires table entries to be written within a tight time limit. Furthermore, P4 devices are generally not designed to perform real-time, with minimal latency in the order of microseconds, updates [34].

4.3.4 *Inverting* the RPS Algorithm

Until now, the control plane was heavily involved in the P4 implementation of RPS; however, in the design goals, we stated that we want to provide an implementation with minimal dependencies on the control plane to reduce the bandwidth overhead. The tight coupling was caused by the

control plane's task to track the mapping of *flowids* to buffer indices, and it can only be broken if the data plane performs its buffer management. Hence, we present a FIFO buffer approach that replaces the buffer with the ability to perform arbitrary modifications as a tradeoff to the ability to perform more operations in the data plane.

The FIFO algorithm In the naive implementation, the control plane managed the *flowid* to buffer index mapping, which was not feasible to do in the data plane due to the unbounded number of receipts per flow. The FIFO implementation circumvents this by ignoring the positions of packets and only looks at the oldest entry in the buffer when replacing an entry. Hence, instead of firstly processing all packets affected by a direct disclosure, packets are removed in the order of arrival from the buffer. The algorithm is presented as pseudo-code in Algorithm 3.

For every incoming packet, a receipt is created and based on the observation if the receipt is a direct disclosure or not, a different path is taken:

Direct disclosure Firstly, the receipt is emitted to the collector identical to the RPS Algorithm and then added to the disclosure tracker to be matched upon deleting an entry from the buffer.

Other In this case, the receipt was not a direct disclosure, and we have to replace the oldest entry, i.e., the head of the FIFO buffer, with the current receipt. The removed entry R is then used to check if a disclosure is present in the tracker for this flow and time range. If this is the case, the conditions for a delayed disclosure are checked identically to the reference RPS Algorithm; otherwise, the receipt is discarded.

This approach is an *inversion* of the RPS Algorithm, as instead of checking for delayed disclosures in the receipts upon a direct disclosure, the removal of a receipt causes a lookup for a direct disclosure to determine if the receipt is a delayed disclosure.

Algorithm 3 Retroactive Packet Sampling Algorithm with a FIFO buffer

1: **procedure** RETROACTIVESAMPLINGFIFO(p : packet)
2: $R' \leftarrow Receipt(p, currentTime)$
3: $r \leftarrow packetRate(R'.flowID)'$
4: **if** $DiscHash(immutable(p)) \in DiscRange(r)$ **then**
5: Emit receipt R'.
6: Add R' to the disclosure tracker
7: **else**
8: $R \leftarrow$ swap the oldest receipt in the FIFO buffer with R'
9: $D \leftarrow$ receipt from the disclosure tracker corresponding to R
10: **if** $\exists D$ **then**
11: **if** $(D.time - R.time) \leq \kappa - \mu$ **then**
12: **continue**
13: **end if**
14: **if** $Hash(R.digest, D.digest) \in Range$ **then**
15: Emit receipt R.
16: **end if**
17: **end if**
18: **if** $LateDisclosure(R.flowID)$ **then**
19: Emit warning.
20: **end if**
21: **end if**
22: **end procedure**

Rate tracking in P4 The RPS Algorithm uses the flow's rate to set the hash range for the direct disclosures. In P4, this can be implemented either as a `direct counter` or `direct meter` associated with a table that is indexed by the *flowid*.

Meter The meter maps the rate using a token bucket to a 2 bit value, which can be used to determine the range based either on hard-coded values or an additional table lookup with the *flowid* and meter colour.

Counter Upon matching the *flowid*, the counter is incremented and the stored rate is returned. The *RPS paper* introduces this approach and allows for an arbitrary number of rate levels as the new rate; thus, the corresponding rate is calculated upon receiving a new direct disclosure. As the value is updated by the controller, a history of rates can be kept and used to smooth out traffic bursts.

Although P4 supports meters out of the box, the implementations use the counter approach due to the simplicity of configuration and monitoring capabilities. Additionally, the counter approach allows replaying traces at different speeds with dedicated timestamps in the packet header.

Disclosure tracker in P4 Compared to the reference RPS Algorithm, which immediately removes all receipts of a flow whilst determining the delayed disclosures, there can now be multiple disclosures active at the same time int the FIFO RPS Algorithm and hence the mapping from an evicted receipt depends not only on the *flowid* but also on the *timestamp* as displayed on a small example in Figure 4.5. The disclosure tracker is modelled in P4 as a table, and the controller

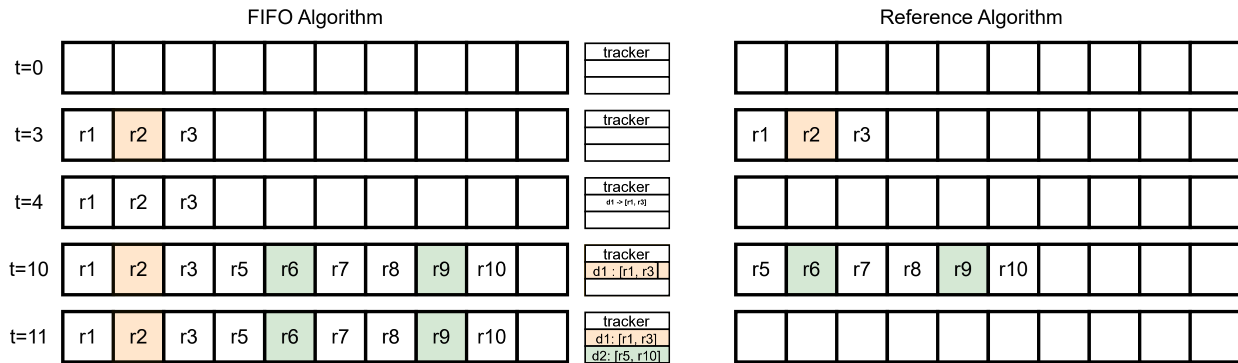


Figure 4.5: Difference in the behaviour of the FIFO RPS algorithm compared to the reference RPS Algorithm for a single flow. A direct disclosure happens for r_4 at $t = 4$ (orange) and r_{11} at $t = 11$ (green). r_2 is a delayed disclosure with respect to d_1 and r_6 and r_9 are delayed disclosures with respect to d_2. Whilst the reference buffer contains only receipts belonging to a single disclosure due to the direct removal, the FIFO buffer can contain receipts belonging to multiple disclosures, as the entries in the tracker indicate.

adds new entries; thus, lookups are efficient. Furthermore, this table also provides the *digest* and *timestamp* of the associated direct disclosure.

There are 2 different approaches which can be used to find the associated disclosure in the tracker, in addition to the *flowid*:

Time This approach uses the `range` match type to find the correct disclosure. A receipt r_x removed from the buffer will have a timestamp in the range $[d_{y-1}.ts, d_y.ts]$ for an associated disclosure d_y. Due to the retroactive sampling aspect, a candidate for a delayed disclosure has to arrive before the associated direct disclosure, and the lower bound is the previous disclosure or 0 if none exists yet.

Number In all examples shown in this work, we number the direct disclosures with incrementing numbers, such as d_1. This numbering can also be added in the data plane as firstly, the direct disclosure range boundary for the direct disclosure must be determined based on the *flowid*. Additionally, we can return the range and the current disclosure number. Then, every receipt stores this following disclosure number and the 12 bytes. Upon evicting the receipt from the buffer, the number can be used as an `exact` key in addition to the *flowid* to resolve the associated direct disclosure. This removes the need for performing `range`-based queries on timestamps at the cost of an additional byte of memory.

These approaches have advantages and disadvantages, discussed later in Section 4.4.5 and Section 5.1.3.

Detecting late disclosure in P4 As detecting late disclosure is a powerful tool when reporting disclosures or their absence to the monitor, an implementation without a `for` loop to determine the oldest receipt has to be found. To accomplish this, a similar approach as in the reference implementation can be made: direct disclosures are added to the buffer. However, their *digest*'s and *timestamp*'s bits are all set to 0 or 1. Combining these two features results in a low probability of a receipt having the same values, thus allowing for good late disclosure detection. Dropping a disclosure from the buffer results in a notification to the control plane, which will store it.

Although an additional table could be used to find the subsequent receipt that is present as soon a direct disclosure occurs, it is advantageous to only point towards the subsequent delayed or direct disclosure for two reasons. Firstly, the control plane has to be notified, and the entry has to be written. At high flow rates, a disclosure could already have happened, thus rendering the efforts useless. Secondly, substantial effort has to be made to remove the table entry when the oldest receipt is found. Hence, the late disclosure warning does not directly point to the following receipt at the time of the direct disclosure as in the RPS Algorithm and thus adds some imprecision. Nevertheless, this is not considered a negative impact on the overall performance, as the monitor only receives disclosures and 'missing' receipts without disclosures do not influence the metric verification. However, a detailed analysis of the monitor with a standardised infrastructure would be needed to understand the detailed impact.

Tradeoffs The advantage of this approach is that the disclosure tracker can be managed by the control plane and thus the controller only interacts upon a direct disclosure with the data plane. However, the drawback is that packets are now removed in the order of arrival and not per flow, which leads to a different buffering behaviour than the reference RPS Algorithm.

4.4 RPS on the Intel® Tofino™

To achieve the goal of providing a production-ready implementation, the generic P4 implementation has to be adapted for an industry-used hardware target. This section introduces the challenges and design adaptations encountered while adapting the implementation to the Tofino™ ASIC. The main challenges were the stage, memory limitations, and time tracking using the 2 bytes *timestamp*.

4.4.1 Stages

The Tofino™ allows for 12 stages of *MAUs* to implement the data plane logic in both the *ingress* and *egress* pipeline. The critical path length requires splitting the algorithm into both pipelines and adding metadata that must be transferred through the traffic manager. As an incoming packet triggers a buffer replacement and the *egress* processing depends on the queue admission, receipts can be dropped.

The compiler is in charge of placing actions and calls to `externs` optimally. However, large tables that occupy multiple stages or dependencies that enlarge the critical path pose a limit to the optimizations. Independent of the implementation choice, the maximization of the buffer space causes the critical path to extend to around 15 stages and thus requires the splitting into the *ingress* and *egress* pipelines whilst the receipt is transferred as metadata.

Under a regular operation regime, there are no drawbacks to splitting the processing. However, due to the queue manager's traffic-shaping policies, the receipt can be lost with the associated packet under heavy loads or congestion to specific output ports. As the receipt is taken from the buffer and thus is independent of the incoming packet, the loss of this potential disclosure can be caused by the internal buffer policy, which is not incorporated in the RPS design.

4.4.2 Storage

The receipt buffer, in combination with the disclosure strategy, is the key part of the RPS Algorithm. As seen in the RPS parameter analysis (Section 2.5), memory in the order of a few Megabytes

is required. The Tofino™ supports up to 4 MB of receipt storage per stage and pipeline under perfect utilization.

As seen before, the only viable solution for storing receipts is the usage of the *register* `extern` in P4. These registers are located in the SRAM and are organized in blocks of 1K entries with 128 bits each. A *MAU* supports up to 48 blocks of registers, with the largest continuous register consisting of 35 blocks. The architecture furthermore limits the sharing of the 128 bit entries and thus allows only for $1, 8, 16, 32, 64$ bit wide entries. With these restrictions and splitting a receipt into 8 bit chunks, storage for up to $573'465$ receipts (4 MB) can be achieved, with the remaining 12 blocks[2] remaining usable by other applications. This block restriction leads to the fact that it is possible to store 4 MB of 7 bytes large receipts ($7 \cdot 8$ bits) and only 3.4 MB of 12 bytes receipts, as the 12 stage limitation requires the usage of 16 bit wide registers, to remain in the order of 6 memory stages. Thus, the $286'733$ 16 bit registers form a bottleneck with 12 bytes $\cdot 286'733 = 3.4$ MB of storage.

Recirculation Although the possibility of implementing the buffer using the recirculation feature exists, no efforts were made towards it as the buffering and additional packet processing does not fit the buffer scenario. The Tofino™ platforms allow for re-circulating packets, and thus, the 22 MB unified packet buffer seems at first glance a better option than storing only 4 MB of receipts. Subtracting a minimal header or even a full *Ethernet* header and padding out the rest of the minimal frame size with receipts, 20 MB or 17 MB of receipt storage could be available in the shared packet buffer. But in the context of the 8 or 16 MB of total receipt storage, assuming all 2 or 4 parallel switch pipelines are used, the trade-offs of sharing the memory with all other packets as well as having the overhead of administrating an equal load balancing and maintaining order, make packet circulation not a viable approach on the Tofino™.

4.4.3 Controller communication

The data plane sends information to the controller if a direct disclosure, delayed disclosure or late disclosure warning is detected. The direct disclosures are time-critical, as the information for the delayed disclosure must be made ready as soon as possible, since the oldest packets are first removed from the buffer. The implementations communicate based on a mixture of polling registers and packet mirroring.

Tradeoffs Many usages of the Tofino™ require only interaction from the control plane to the data plane for configuration, and registers are periodically read for the control plane to track the state of the data plane. The data plane can use the *LearningFilters* to send small message digests to the controller to trigger state updates or write entries into registers. The former approach relies on the ASIC for the batching of the requests, while the latter requires regular polling. An alternative is the usage of packet mirroring, which creates a copy of a packet and allows for header modification such that the desired data is sent to the controller.

Whilst the learning filter, which is only available during the ingress processing on the Tofino™, is the ideal abstraction for sending data to the controller, the time between packet reception and data arrival in the controller depends on the Tofino™ ASIC. For the time-critical direct disclosures, the digest feature was found to have a larger transmission time than transmitting a dedicated message over *ethernet*. The cause for this larger delay is not directly due to the transmission

[2] A register requires an additional block for the access synchronization, thus requiring 36 of 48 blocks.

channel. With every $\approx 10^5$-th packet being a disclosure, the internal digest buffers are not filled fast enough, and a time latch forces the exchange based on the ASIC configuration.

The transmission over the network depends on two factors: the Tofino™ delay and the BMC delay. The delay on the Tofino™ is the time in the traffic manager and the egress processing, while the time on the BMC consists of receiving the packet and sending a signal to the controller. The former is expected to happen within hundreds of nanoseconds, while the latter can be in the order of a few microseconds [35]. The mirroring of packets reduces the overall forwarding rate, however, the impact is comparable to other sampling schemes such as with NetFlow [36] due to the selection of σ.

Using registers avoids dependency on the network and thus avoids potential packet loss. Disclosures are stored in a register array read from the control plane. Accessing a register for a read operation on the Tofino™ takes an average of 1 to 5 us based on the reporting of Intel®'s benchmarking script. A tracking register can determine the indices in a dedicated direct disclosure buffer to read out, allowing for batching of the requests. Whilst the previous communication primitives don't use SRAM, this approach uses SRAM to store the register contents.

Used primitives The provided final implementation uses a mixture of register polling and mirroring. The direct disclosures are shared over a 4′000 entries large register array, requiring 35 KB of register memory for the optimized receipts. Mirroring is used for delayed disclosures and late disclosure warnings that are not time-critical. With a rate of 1 Gpps and a disclosure rate $delta$ in the order of 10^{-5}, about 10′000 disclosures can be expected per second. To have some headroom, 4′000 entries are selected to allow also consistent readouts during traffic bursts.

4.4.4 Timestamps

As introduced, the reference implementation from the *RPS paper* uses 2 bytes to store the timestamp at millisecond accuracy. As the Tofino™ lacks the support for general division and modulo operations, special care with the time calculations has to be taken. This includes changing time resolution and dedicated overflow handling.

Extracting timestamps A global timestamp with *ns* accuracy is generated during the ingress parsing. To get a subsection of this timestamp in *ms*, a division by 1′000′000 would be required. Although the Tofino™ provides a MathUnit `extern` that offers divisions, the operation can only be added to a register, which is impractical due to calculating the timestamp for every packet. Hence, the timestamp is reduced by shifting 20 bits, resulting in a resolution of 1′048′575 *ns*. Using this scaling, the 2 bytes timestamp resolves to a maximal duration of 68 seconds.

Determining time differences With the two approaches, different time resolutions are needed. If the *timestamp* is used as a key in the *range* key for the lookup, the complexity is pushed to the controller to provide the correct values. In the case of the *number*, the data plane has to resolve the timestamps, which is done with the aid of the control plane.

With the limited operations outside of stateful ALUs, especially the restriction of operations involving at most 4 bytes of data and 12 bits of PHV info, the time difference cannot be directly computed. Therefore, the controller provides the *timestamp* of the disclosure d, the earliest timestamp before the disclosure ts_q based on the quiet time $(\kappa - \mu)$ and a flag, indicating if an overflow occurred in the computation. Using this information, the data plane determines

whether the *timestamp* of a receipt r falls into the quiet time. Hence, to perform the check $(d.ts - r.ts) \leq (\kappa - \mu)$, the P4 code as shown in Listing 4.1 is used:

```
bit<16> greater_q = max<bit<16>>(receipt.ts, disclosure.ts_with_quiet_time);
bit<16> smaller_d = min<bit<16>>(receipt.ts, disclosure.ts);
bool above_q = greater_q == receipt.ts;
bool below_d = smaller_d == receipt.ts;

if (disclosure.ts_overflow){
    if (above_q || below_d){
        // do nothing
    } else{
        delayed_disclosure();
    }
} else{
    if (above_q && below_d){
        // do nothing
    } else{
        delayed_disclosure();
    }
}
```

Listing 4.1: P4 case distinction to determine if a receipt's timestamp falls into the quiet time.

Noteworthy is the combination of lines 1 and 3: With the P4 limitations, $a \geq b$ cannot be directly computed and thus, it is checked by $max(a, b) == a$. The lines 13 to 17 handle the case without overflow by checking if $ts_q \leq r.ts \leq d.ts$. In the case of an overflow, $d.ts < ts_q$ and thus, the check is split into $ts_q \leq r.ts \vee r.ts \leq d.ts$.

4.4.5 Final implementation

Whilst the usage of the time range initially seems to be a better choice than the approach with an explicit *next disclosure number* $(next_d)$, the efforts in table management and state mirroring to the control plane lead to the decision to use an additional byte per receipt to store the per-flow next disclosure number. This number is incremented for every collected direct disclosure such that for 2 direct disclosures d_i and d_j, $next_d(d_i) = next_d(d_j) + 1$. Hence, with the rate update of d_i, the next disclosure number is also incremented, and every consecutive receipt is tagged with this number. Removed receipts from the buffer use the *flowid* and $(next_d)$ to find the corresponding direct disclosure for the delayed disclosure. This approach, on the one hand, removes the non-trivial time-range calculation in the disclosure table, as the $(next_d)$ is a sequential number that can easily be tracked. For example, upon overflowing the *timestamp*, two dedicated entries must be written as the range requires $low \leq high$. On the other hand, the *range* match-type is removed, and thus, less table memory is used as range entries are split into multiple entries in the table.

4.4.6 Not considered

While many implementation decisions were made, some challenges were out of the scope of this work and are thus summarized here.

Timestamps The current implementations use the `global_tstamp` provided by the devices' ingress deparser metadata. This approach was chosen due to the Tofino™'s support of the *Precision Timestamp Protocol (PTP)* [37]. However, the API endpoints are only exposed in the C API, and thus, the time synchronization was not used, and the timestamps are directly extracted without adaptation.

Mirroring Depending on the switch built around the Tofino™ ASIC, packet mirroring is not always supported over a dedicated interface to the BMC. Specific examples of this are the Wedge 100BF-32Q and Edgecore Networks WEDGE100BF-64Q. The former has the dual 10 Gbit/s internal network connection to the BMC, while the latter only offers a dedicated 100 Gbit/s port on the front panel. If this interface cannot be connected to the BMC, the *bf-driver* can be switched to send packets over the PCIe interface. The exact configuration is left for future implementations.

Controller The controller is a central piece of the RPS application, but only a proof of concept is provided in Python. To accommodate all required functionalities and access to all Tofino™ features, a *C* implementation should be used. Not only does this offer better feature coverage, but it also allows for a deeper interaction with the underlying hardware.

Chapter 5

Evaluation

The proposed implementations are evaluated in the following three dimensions:

1. Hardware usage compared to an L2-forwarding reference
2. Disclosure accuracy compared to the reference RPS Algorithm using a simulator
3. Performance on the Edgecore Networks WEDGE100BF-32Q

We find that the best implementation using in-data plane buffering uses 11 of the 12 stages as well as 45% of the Map RAM and around 30% of the overall SRAM are consumed, whilst the remaining resource usage is within 10%. Using a simulator, we find that, the FIFO RPS Algorithm achieves the target accuracies in most cases within the specified time, averaging around 10% less samples than the RPS Algorithm under identical configurations. Implementations running on the Tofino™ can support up to 2.3 Mpps at a minimal flow rate of $r_{min} = 200$ Kpps while maintaining the target accuracy. However, results obtained on the hardware show the need for an optimised controller, as only 1.5% of the generated digest could be received when using the digest mode.

5.1 Hardware resource usage

As AS operators should be incentivised to deploy RPS in their networks with minimal additional effort, the resource usage of the implementation is a crucial aspect. Ideally, the number of stages available, TCAM and SRAM, is barely reduced. We find that all implementations with an in-data plane buffer use around 10 stages and consume up to 50% of the available SRAM per stage for up to 8 stages (30% average over all stages), and thus block a significant portion of the available SRAM. Furthermore, the Map RAM is intensely used (up to 50%), while most other usage metrics fall below 10%. Implementations without an in-data plane buffer use up to 2 stages and, at most, 4% of the available resources.

5.1.1 Methodology

Environment All implementations are compiled in a virtual environment based on Ubuntu 20.04 LTS with the Intel® P4 Studio SDE 9.13.1. The compilation is initialised over Python using *p4utils* [38]. The Intel® P4 compiler generates all hardware usage numbers reported, and additional insights are obtained using Intel®'s *p4Insight*.

Reference A custom-written P4 program called `l2 switch` is used as a reference, which consists of a single table that performs the modification of the *Ethernet* header and is a fundamental building block for all other implementations. It ensures the functionality of the switch as a network switch, although MAC learning is not implemented, and all entries have to be pre-configured from the control plane.

Parameters The evaluation focuses on the resource usage of the ingress and egress pipeline; thus, the parser is not considered. The dissection into *Ethernet*, *IP*, *TCP* or *UDP* headers is a common use-case as well as it is identical among all tested implementations. Hence, the usage of the number of match action stages and the limited SRAM and TCAM are evaluated. A third, Tofino™ specific, memory is the MAP RAM, which is used for stateful objects such as counters or meters. A further metric is the usage of the Match Crossbar, which is responsible for providing the search keys and the action selection; a high use limits other applications from performing table operations [39]. The very *long instruction word (VLIW)* actions are the instructions processed in a clock cycle in the switch, and the hash bits are the number of bits used internally to determine table entries. The last parameter reported is the number of *Stateful Arithmetic Logic Units (ALUs)* used. These units can perform atomic updates for `extern` functions.

5.1.2 Implementations

The following six implementations are compared against each other and the *Reference*. The buffer size is dimensioned so the source code compiles for the Tofino™ target.

Digest The *Digest* implementation uses the *LearningFilter* to transmit the 12 bytes receipts to the controller, as introduced in Section 4.2.2.

Network This implementation uses the mirror operation to send the first 48 immutable header bytes and the 2 bytes wide timestamp to the controller over the *Ethernet*, as introduced in Section 4.2.1.

Naive The *Naive* implementation uses the controller's instructions but buffers the receipts in the data plane as introduced in Section 4.3.3.

FIFO The *FIFO* implementation tracks the disclosures and stores the receipts in the data plane. The implementation was introduced in Section 4.3.4.

FIFO Opt This implementation uses the *FIFO* implementation, but reduces the *flowid* to a single byte as described in Section 4.4.2.

FIFO Note The *FIFO Note* implementation uses an additional byte to track the *next disclosure number*. Thus, it does not rely on the timestamp to find the associated disclosure. The implementation was introduced in Section 4.4.5.

5.1.3 Results

Resource	Reference	Digest	Network	Naive	FIFO	FIFO Opt	FIFO Note
Stages	1	2	1	11	10	11	10
Match Crossbar	6	72	11	132	205	153	156
Map RAM	0	0	0	256	256	256	292
SRAM	4	4	9	268	279	278	318
TCAM	0	0	0	0	75	25	0
VLIW Actions	2	7	4	20	19	21	19
Hash Bits	40	104	80	361	393	409	472
Stateful ALUs	0	0	0	1	1	1	1

Table 5.1: Absolute resource usage summed over all stages of various P4 implementations on the Tofino™.

Resource	Reference	Digest	Network	Naive	FIFO	FIFO Opt	FIFO Note
Stages	8.33%	16.67%	8.33%	91.67%	83.33%	91.67%	83.33%
Match Crossbar	0.26%	3.09%	0.47%	5.67%	8.81%	6.57%	6.70%
Map RAM	0%	0%	0%	44.44%	44.44%	44.44%	50.69%
SRAM	0.42%	0.42%	0.94%	27.92%	29.06%	28.96%	33.12%
TCAM	0%	0%	0%	0%	26.04%	8.68%	0%
VLIW Actions	0.52%	1.82%	1.04%	5.21%	4.95%	5.47%	4.95%
Hash Bits	0.80%	2.08%	1.60%	7.23%	7.87%	8.19%	9.46%
Stateful ALUs	0%	0%	0%	2.08%	2.08%	2.08%	2.08%

Table 5.2: Utilisation of the available resources on Tofino™. The percentage is the ratio between the used resources summed over all stages divided by the available resources over all stages.

Stages All implementations with a register-based buffer (*Naive*, *FIFO*, *FIFO Opt*, *FIFO Note*) use nearly all but 1 stage of the 12 available stages (Table 5.1). This high usage is due to two factors: the dependencies of the critical path and the register placement favouring the largest possible sizes. The dependencies force the usage of 3 stages as the direct disclosure threshold needs to be fetched, and the next buffer index needs to be determined. For the second part, 7 to 8 stages in the *Ingress* pipeline are reserved for pure buffering (*buffer stages*), as introduced in Section 4.4.2. Although the *FIFO Opt* has one less stage filled with registers due to not having the next disclosure number, *FIFO Note* is compiled with one less stage, which is unintuitive. Based on the compiler output, the *FIFO Note* parallelises the digest calculation and disclosure threshold resolving, which is not done in other implementations. The last stage of the *FIFO Opt* contains a single action. This action is executed in a previous stage in the *FIFO Note*. The exact reason for these optimisations is unknown. Due to the simple operations, the *Reference* and *Network* only use a single stage. The *Digest* implementation uses an additional stage to calculate the *digest*.

SRAM In addition to the high stage utilization, the *SRAM* for the registers utilized around 30% (Table 5.2) of the available SRAM. Although 30% might not directly seem high, this value is reported over all stages. The first 3 stages' utilisation is near 0%, whilst the buffer stages (7 to 8 stages) are utilised at 45% each. *Naive*, *FIFO*, and *FIFO Opt* have nearly identical SRAM usage (+-1%) due to a similar amount of buffering stages and disclosure look-up table entries. *FIFO Note*

stores one additional byte per receipt and thus requires more SRAM. *Reference*, *Network* and *Digest* use only very little SRAM ($< 1\%$) due to the few tables. The *Network* has an additional table for resolving the destination for the receipts and thus has a slightly (0.5%) higher SRAM usage.

Map RAM All implementations with a register-based buffer (*Naive*, *FIFO*, *FIFO Opt*, *FIFO Note*) use 45% of the Map RAM to support the registers, with the buffer stages having a utilisation of 75%. Leaving only 25% of the resources available for other meters, counters or registers or range entries with ageing. All implementations without registers and counters, namely *Reference*, *Network* and *Digest* don't utilize the Map RAM.

TCAM Due to the usage of the range match type, *FIFO* and *FIFO Opt* utilize up to 25% of the available TCAM. More precisely, *FIFO* utilises the TCAM in the first stage 100% and *FIFO* fully utilises the first 3 stages for determining the disclosure with the key $< \textit{flowid}, \textit{timestamp} >$. As the *flowid* in the *FIFO Opt* is smaller than in the *FIFO*, less TCAM is required. None of the other implementations use TCAM due to avoiding ternary matches.

Match Crossbar The more complex implementations have more actions and conditional, thus containing more instructions and requiring more data inputs from the Match Crossbar. This can be observed by comparing the *Reference*, *Network* and *Digest* implementations. As *Digest* requires access to the data field to calculate the digest hash, the Match Crossbar usage increases by 2%. The more complex remaining implementations use around 5% of the available Match Crossbar. Their usage is similar due to the comparable data access patterns.

VLIW Actions Similarly to the Crossbar usage, the more instruction an implementation contains, the more VLIW Actions are used, up to a maximum of 5.5% for the *FIFO* implementation.

Hash Bits As the hash bits are used to index tables with the match type `exact`, which also includes registers, the hash bit usage increases with more complex and more extensive implementations. The highest usage is reported for *FIFO Note*, using 9.5% of the available hash bits, as it contains the most registers.

Stateful ALUs A single Stateful ALU is required for the buffer-based implementations, as the *buffer index* has to be increased atomically, thus requiring 1 Stateful ALU. Implementations without a data plane buffer for receipts don't use a Stateful ALU.

5.2 Accuracy

Accuracy is critical when deploying RPS, as the monitor uses missing receipts to attribute packet loss. Hence, every implementation must report missing packets correctly due to memory constraints. Furthermore, a high percentage of the disclosures compared to the reference RPS Algorithm needs to be collected to satisfy the monitor's target number of sample requirements. We show the absolute number of collected samples, the number of disclosures, and the additional time required to collect the target sample count compared to the set collection time of $T = 10$ minutes.

5.2.1 Simulation Setup

To analyse the impacts of different RPS implementations, we created a simulator to reliably and accurately replay packet traces without interference that occurs due to packet buffering or memory contention. Its input is an experiment configuration consisting of three sub-configurations that specify the entire experiment.

1. The trace configuration defines how multiple CAIDA traces are merged into a single `.pcap` trace. This also includes options such as adding debug headers or reducing the packet to a header-only format.

2. The algorithm parameters are the assumptions about the trace's characteristics, such as the r_{min} and r_{max}, as well as the fixed parameters, such as σ. For transparency, the buffer ratio and buffer size are specified so as not to rely on the parameter solver.

3. The simulator supports various implementation types, each with multiple configuration options. The implementation configuration specifies which parameters are used on which instance of an implementation.

All three configuration parts are grouped into a single configuration file and passed to the simulator as input. The simulator is responsible for parsing the `.pcap` trace and provide the resulting receipt to every implementation, as well as the collection of the generated logging artefacts, including the disclosures (Figure 5.1).

Simulation scope The simulator abstracts the hardware to focus on the implementation under ideal conditions. In the context of the RPS implementation, this means that the processing of a receipt and all effects caused by it are fully visible upon generating the following receipt unless the implementation explicitly adds a delay. This assumption is acceptable for packet rates until the Mpps range, as inter-packet times remain in the order of 1 *us*. However, at rates around Bpps, only 1 ns between packets remain, and the implementation must be scaled out to keep the line rate. With average packet sizes of 800 bytes in the CAIDA traces and Tier-1 AS such as *Cogent* having an inter-city capacity of up to 6 Tbit/s [40], a full utilisation would reach 1.5 Bpps. During normal operations, these links are, on the one hand, rarely fully saturated. On the other hand, this traffic rate is not necessarily transmitted to neighbouring AS at a single location.

5.2.2 Methodology

In the experiments, two different datasets simulate the traffic behaviour an ISP might encounter. The first trace type is referred to as *balanced*, while the second is referred to as *imbalanced*. All datasets consist of multiple CAIDA [13] Traces are combined to produce realistic inter-packet arrival times and packet headers. As network traffic differs daily and from hour to hour, every dataset contains 6 combined traces to mimic the variability. A full overview of the used traces can be found in Appendix B.

Balanced The *balanced* dataset consists of 6 combined traces, each composed of multiple CAIDA traces interpreted as a single flow. Each source file is recorded on a different day, such that the up to 8 flows combined into a trace have variabilities. Each combined trace in the dataset is replayed with increasing flows from 1 to 8, increasing by powers of two. Every flow has a minimal rate of 400 Kpps and a maximal rate of approximately 700 Kpps (Figure 5.2), corresponding to an average

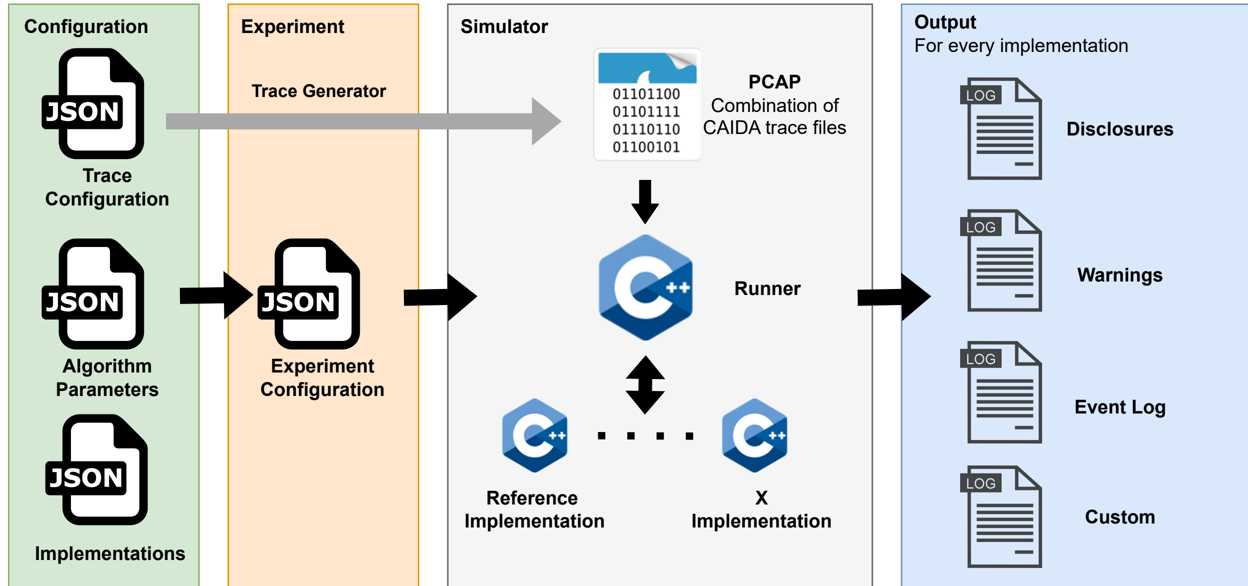


Figure 5.1: Simulation workflow. The experiment setup and the used parameters are encoded in a JSON file format to facilitate fast configurations and easy replicability. The simulation runner replays a pre-computed trace and produces per-implementation artefacts, such as the list of all disclosures.

rate per flow of 3.5 Gbit/s with an average packet size of 800 bytes. As every flow has a similar rate, the combined trace is called *balanced*. This traffic type abstracts the operational mode under normal conditions where a different ISP might have more flows, but the traffic share remains constant. With increasing the number of flows, the scaling behaviour under similar conditions can be analysed.

Imbalanced In contrast to the *balanced* dataset, the *imbalanced* dataset consists of 6 combined traces, each containing a single flow collocated with a higher rate flow. We model this by shifting multiple traces of a single day into a larger flow. Moving together n traces, we obtain an amplification of n and thus model an imbalance of around $1 : n$ (Figure 5.3). Each combined trace in the dataset is replayed with increasing imbalance from 1:1 till 1:8, increasing by powers of two. The factor is only approximately $1 : n$ as traffic rates vary over time. To keep a realistic behaviour, all source traces for the large flow are sourced from the same collection day. This scenario is used to understand the impact of allocating multiple flows.

RPS Parameter Configuration For a deployed algorithm, the default collection time is $T = 10$ min. To keep the file sizes manageable, the constraint of simulating only a single minute of traffic reduces the expected number of samples to $N_{\gamma,\varepsilon}/T$. To compare the collected disclosure number to $N_{\gamma,\varepsilon}$, the number is scaled linearly as if the experiment ran for T minutes. Furthermore, we use $r_{min} = 200$ Kpps, $r_{max} = 700$ Kpps, $loss_{min} = 0.1\%$, $\varepsilon = 10\%$, $\gamma = 95\%$, $\kappa = 100$ ms, $\sigma = 1\%$, $\delta_{high_rate} = 1.37 \cdot 10^{-6}$, $\delta_{low_rate} = 2.185 \cdot 10^{-5}$ and a *threshold* for switching between δ_{high_rate} and δ_{low_rate} at 437 Kpps. The R^* is scaled by the number of flows. All parameters were chosen due to their usage in the RPS paper. As R^* is scaled, the buffer size is scaled by the optimised buffer ratio of ≈ 0.24. With this buffer ratio, the non-optimised Tofino™ can fit a maximal of around 3.4 MB, whilst the optimised increases this limit to 4 MB (Figure 5.4) due to the reasons introduced

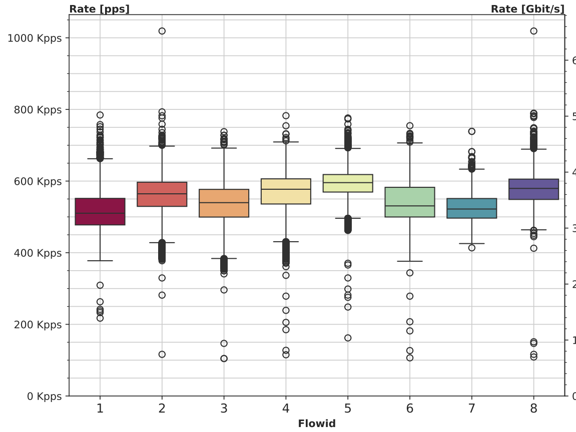


Figure 5.2: Per-flow rate of the *balanced* dataset, sampled every 50 *ms*.

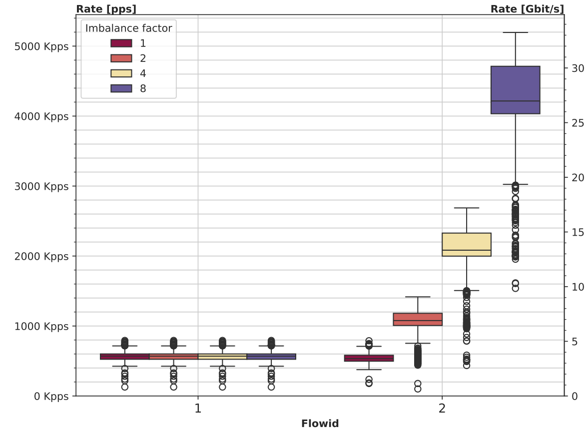


Figure 5.3: Per-flow rate of the *imbalanced* dataset, sampled every 50 *ms*.

in earlier sections (Section 4.4.2). This limitation results under the ideal disclosure strategy in a peak handlable traffic capacity R^* of up to 15 Gbit/s per pipeline[1]. Similarly, Figure 5.5 shows R^* as packet rate per second. Assuming the worst case of minimal-sized packets, the Tofino™ can handle up to 1.22 Gbit/s traffic under ideal disclosure conditions per pipeline.

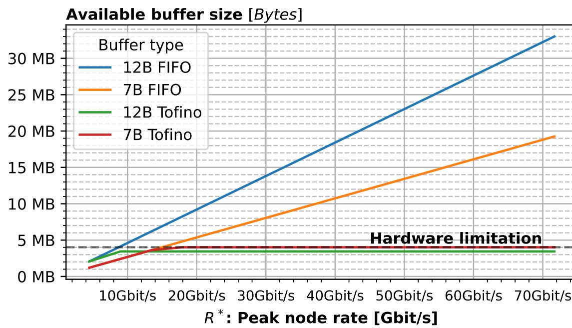


Figure 5.4: Buffer size in relation to the peak rate at the switch for a single pipeline, using an average packet size of 800 bytes at r_{min} = 200 Kpps.

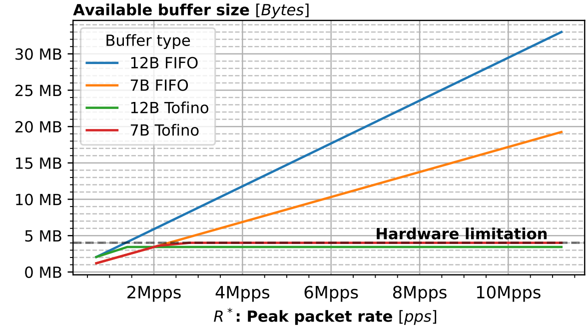


Figure 5.5: Buffer size in relation to the peak rate at the switch for a single pipeline at r_{min} = 200 Kpps.

Reference To provide a compatible implementation of the RPS Algorithm presented in the *RPS paper*, all simulated implementations are compared to the *Reference* implementation introduced in Section 2.6. Furthermore, an *Ideal* implementation that uses the RPS Algorithm but with an infinite buffer was used to compute the superset of all disclosures to ensure the validity of the generated receipts of all implementations. In Appendix C, the *Ideal* and the *Reference* are compared. The *Reference* uses the disclosure strategy based on the RPS Algorithm and has a finite buffer based on the previously presented algorithm parametrisation.

[1] The Tofino™ has 2 or 4 independent pipelines based on the model

5.2.3 Implementations

In this part of the evaluation, only the implementations that generate the receipts and buffer them in the data plane are evaluated. As the interaction between data plane and control plane is needed, we also provide *delay* implementations that model the delay until a value written by the control plane is visible in the data plane. Research indicates that adding a table entry induces overheads in the ideal case of around 26 *us* [41]. Although this value is only measured for inserting a table entry, it is used as a reference for sending a signal to the controller and seeing the effect in the data plane.

The following six implementations are considered due to the different buffer sizes:

FIFO This implementation uses the FIFO RPS Algorithm as presented in in the previous chapter.

delayedFIFO The implementation uses the FIFO RPS Algorithm as well as it adds a delay of 26 us for data plane table modifications.

Tofino This implementation extends the *FIFO* implementation but limits the buffer size to 286K buffer slots to store the 12 bytes wide receipt.

delayedTofino Extends the *Tofino* implementation by adding a delay of 26 *us* for modifying data plane table entries.

TofinoOpt This implementation extends the *FIFO* implementation but limits the buffer size to the optimal case of 573K buffer slots. A receipt consists of a 1 byte long *flowid*, 2 bytes for the *timestamp* and 4 bytes for the *digest*. The number of flows is reduced to 256, and the mapping is assumed to be predefined.

delayed TofinoOpt Extends the *TofinoOpt* implementation by adding a delay of 26 *us* for modifying data plane table entries.

5.2.4 Overview

Before diving into an implementation-specific analysis, an overview of 6 different implementations is provided. Combining all experiments run, mixing *balanced* and *imbalanced* datasets, the *Ideal*, *Reference* and *FIFO* sample more than the lower limit of $N_{\gamma=95\%,\varepsilon=10\%}$ and thus achieve the target sample count before the defined collection time is reached (Figure 5.6). On the other hand, the *Tofino* barely achieves the target collection time on average but requires over an hour in the worst case. Hence, it seems that the *FIFO* can be comparable to the *Reference*, whilst the *Tofino* only achieves this in some instances.

All implementations, except for the ones with the reduced buffer capacity due to the Tofino™'s memory constraints (*Tofino*, *delayedTofino*), achieve an error rate ε smaller than the specified 10% in 75% off all cases in both datasets, with only the *Ideal* implementation achieving an error level always below 10% (Figure 5.7). The optimised implementations for the Tofino™ achieve the target accuracy in around 40% of all cases, with worst-case results in the order of 50% error.

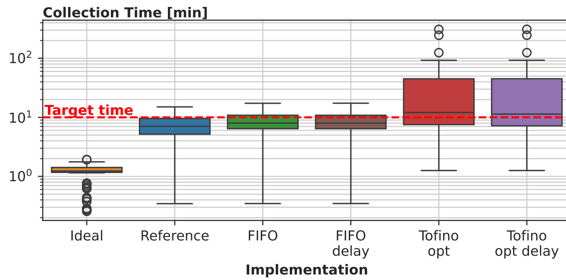


Figure 5.6: Actual collection time when configured to $T = 10$ minutes, considering the per-flow samples over all traces.

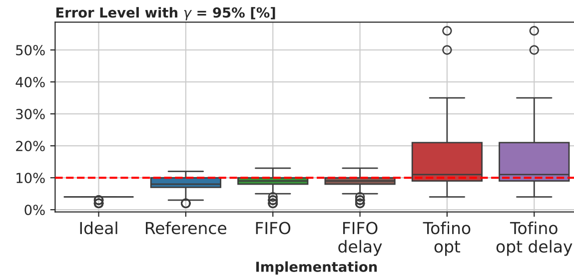


Figure 5.7: Error rate achievable within the collection time $T = 10$ minutes at $\gamma = 95\%$. Configured (target) error rate $\varepsilon = 10\%$.

5.2.5 P4 Reference model

In this phase, the *FIFO* implementations are evaluated against the *Reference*, which uses the ideal disclosure strategy with arbitrary buffer removals. Although the non-optimal disclosure strategy (FIFO RPS Algorithm) is used for the *FIFO* and the *delayedFIFO*, the same target times and thus accuracies, whilst collecting about 10% less receipts can be satisfied compared to the *Reference*. Furthermore, imbalances between flows do not harm the overall sampling performance with correctly scaled buffers under a non-optimal disclosure strategy. Additionally, the overhead of 26 *us* has a negligible impact on the sampling performance.

Accuracy Using the *balanced* dataset and evaluating the number of collected samples for the 1 to 8, flows by first starting at a single flow and adding more flows until 8 are reached, *FIFO* and *delayedFIFO* requires about 10% more time to collect the same amount of disclosures as *Reference* (Figure 5.8). Whilst *Reference* collects all samples below 10 *min* on average, the flow 7 takes 6% longer to achieve the same number of samples on average. This is due to the different disclosure strategies: *Reference* uses the FIFO RPS Algorithm that removes receipts instantly upon disclosure and thus frees the buffer, allowing it to compensate for flow rate variabilities. The *FIFO* and *delayedFIFO* use the FIFO RPS Algorithm as their disclosure strategy, and without removing the receipts instantly, rate variabilities have a higher impact, as receipts from a different flow wait for eviction. However, such minor violations can be fixed by slightly increasing the *threshold* of δ. Comparing the worst-case performance, *Reference* requires an additional 20% time to collect the samples whilst *FIFO* and *delayedFIFO* require an additional 40%, twice as much as *Reference*.

Stability To compare the stability of flows when adding additional flows with the *balanced* dataset, the number of flows is increased and the difference in disclosure is evaluated compared to the first flow observation. E.g. flow 1 is first observed with 1 flow; hence, when evaluating 2 flows, the ratio of disclosures $\#d_2/\#d_1$ can be assessed. Ideally, the flows have a ratio above $100\% = 1.0$, indicating that more flows lead to more collected disclosures due to the sharing of unused buffer space.

Reference increases the number of collected samples in 6 out of 7 cases, whilst *FIFO* and *delayedFIFO* only do this in 4 out of 7 cases (Figure 5.9). This can again be attributed to the difference in the disclosure strategy. Comparing the overall spread, all implementations stay within similar bounds, except for flow 1 when collocated with 7 other flows, indicating a comparable

behaviour when increasing the overall traffic and buffer size. All implementations can gain additional disclosures from sharing buffer capacity.

Furthermore, the performance impact of the *delayedFIFO* compared to the *FIFO* is negligible. This is mainly due to using packet rates up to 5 Mpps, offering a time window of around 130 packets in the buffer, for an action to be processed. Combined with the possibility of multiple disclosures being in the buffer, the impacts are insignificant, with a share of the difference around 0.15% compared to disclosures without delay ($\leq$ 100 packets). However, more flows lead to more variability and, thus, to the possibility of a larger impact of the delay.

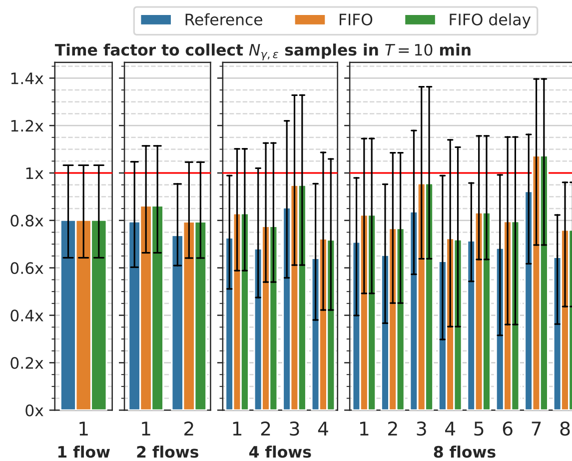


Figure 5.8: Time compared to the baseline of $T = 10$ min to collect the samples using the *balanced* dataset. The bars represent the median, and the error bars represent $[min, max]$.

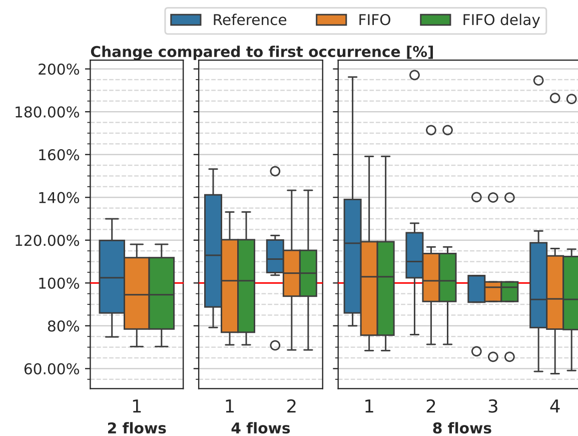


Figure 5.9: Relative change per flow compared to the first occurrence using the *balanced* dataset.

Impact of imbalanced flows Switching to the *imbalanced* dataset and steadily increasing the imbalance, all implementations collect the required samples before the target time runs out. However, when comparing the number of collected samples, a difference can be noticed: Whilst *FIFO* and *delayedFIFO* collect a constant amount of samples, within 90% of the target time, the collection time is nearly halved from 1:1 to 1:8 for *Reference*'s flow 1. But again, this can be explained by the different disclosure schemes and the *threshold* for changing between the disclosure rates δ. With *Reference*, the larger flow removes more receipts from its buffer part because of the fixed *threshold*. After switching the rate, there is no further adaptation and disclosures are very often released such that the overall collection time drops to 10% of the target time (Figure 5.10). While this is happening, the low-rate flow is not evicted from the buffer due to the self-clearing of the high-rate flow. Thus, this low-rate flow can accumulate more receipts for the delayed disclosure. For *FIFO* and *delayedFIFO*, this does not uphold, as the low-rate flow is pushed out by the FIFO queue at the flow's combined rate, making the "growing" of the low-rate flow impossible.

As the buffer size satisfies the requirements, the number of disclosures stays for the small flow within the collection time. Furthermore, when comparing the spread of the *min-max* interval of flow 1, a noticeable difference can be observed between *Reference* and the other two implementations: Whilst the spread is reduced from initially $0.7x$ to $0.25x$, the other two implementations' spread remains nearly constant at around $0.85x$. This, again, can be explained by

the buffering behaviour and the constant removal of receipts when using the FIFO RPS Algorithm.

The second flow's collection time is exponentially reduced, due to the usage of the single *threshold*, showing the need for a more fine-grained disclosure rate control to avoid oversampling.

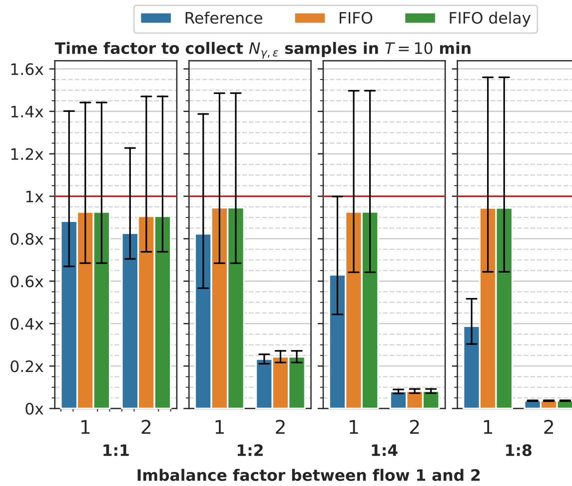


Figure 5.10: Time compared to the baseline of $T = 10$ min to collect the samples with the *imbalanced* dataset. The error bars represent $[min, max]$.

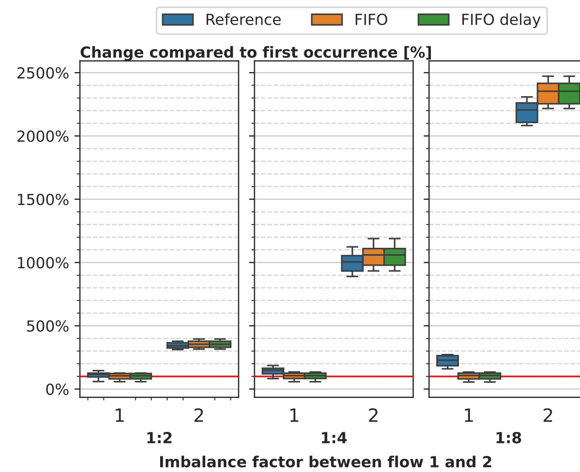


Figure 5.11: Relative change per flow compared to the first occurrence using the *imbalanced* dataset.

Moving on and comparing the change to the first occurrence, the effect of the *threshold* and the buffer sharing can be observed when looking at flow 2 of *Reference* in Figure 5.11. The number of disclosures slightly decreases compared to *FIFO* and *delayedFIFO*. This is due to more disclosures falling into the quiet time, as introduced in Section 2.5.6, because the buffer is clearing faster than intended. The other two implementations are invariant to this effect as they cannot remove entries from the buffer at arbitrary positions.

Collection time variability In both experiments, different intervals of the $[min, max]$ from small to large can be observed, posing the question of how this can be possible as every combined trace in the dataset uses identical parameters (Figure 5.8, Figure 5.10). The observed behaviour is due to the network traffic's burstiness, the linear scaling from a single minute to 10 *min* and the single-state rate tracker used.

The smallest interval was found with a range of 0.05 in the *imbalanced* dataset on flow 2 for the imbalance factor 1:8 and the largest at 1.1 for the *balanced* dataset at 8 flows for flow 3. Translating these ratios into a pure time based on T, they range from 30 *sec* to 11 *min*. With $T = 10$ *min*, requiring nearly twice the collection time to collect the samples is far from ideal. However, the achieved performance was scaled by 10x, amplifying under- or oversampling effects. Such undersampling effects can be caused by traffic bursts, causing more than average receipts to be discarded due to the quiet time. With a burst occurring, the single-state rate-tracker causes an overestimation of the rate and, thus, a lower sampling performance. An optimisation of the rate-tracker could use a moving average for the rate to compensate for this, as it was suggested in the RPS *paper*.

5.2.6 Tofino model

The Tofino™ imposes a strict buffer size limit and thus allows only for a reduced rate as introduced in the RPS parameter configuration (Section 5.2.2). In this section, the implementations on the Tofino™ are exposed to traffic rates overwhelming their specified buffer size.

Hence, these findings serve as an indicator of the behaviour if insufficient buffer size is provided. Overall, the *Tofino* fails to collect the required samples within the target time if more than 2 flows are involved in the *balanced* dataset or the imbalance exceeds 1:2 in the *imbalanced* dataset.

Not enough samples The investigation of the limited buffer size starts by comparing the number of collected samples with increasing flows on the *balanced* dataset. While all 5 implementations collect more than the required number of samples for a single flow, *Tofino* and *delayedTofino* fail to collect the samples for 2 flows ($\approx$ 1 Mpps). With 4 flows, both of these implementations only collect around $1/3$ of the required samples, dropping to near 0 with 8 flows. A similar fate, but only staring at 4 flows ($\approx$ 2 Mpps), is encountered by *TofinoOpt* and *delayedTofinoOpt* (Figure 5.12).

Comparing the change of each flow with an increasing number of flows, the detrimental impact of an under-sized buffer (280K entries instead 1375K required entries) for a given packet rate can be seen when looking at the *min-max* interval of *Tofino* and *delayedTofino* at 8 flows, which only spans around $1K$ packets whilst the optimised implementations with $2x$ the buffer size have a range of around $20K$ entries (Figure 5.13).

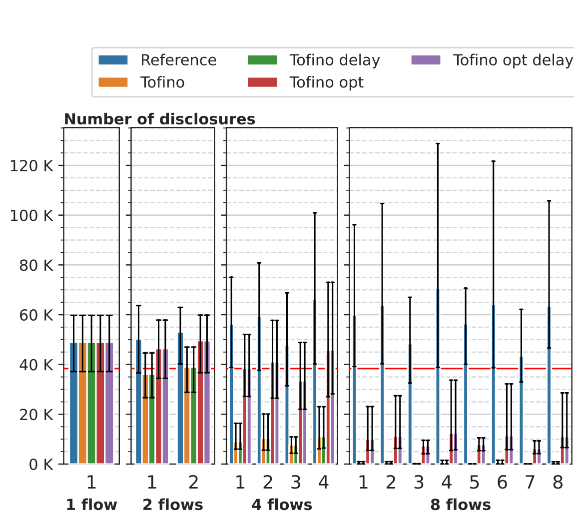


Figure 5.12: Number of samples collected within a minute using the *balanced* dataset. The error bars represent $[min, max]$.

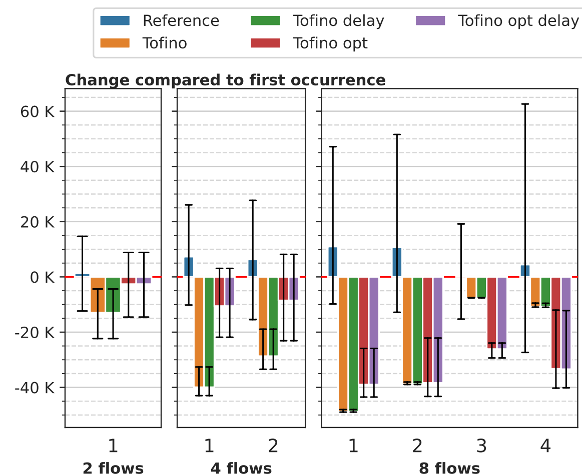


Figure 5.13: Absolute change of the collected samples per flow compared to the first occurrence using the *balanced* dataset.

Victimizing low-rate flows Revisiting the *imbalanced* dataset and comparing the number of disclosures, the results of the *balanced* dataset are repeated for the lower-rate flow with the number 1. The *Tofino* and *delayedTofino* collect enough samples for the 1:2 imbalance, but for all further imbalances fail to collect enough samples due to the limited buffer size and the quiet time rendering samples in the too small, ($5x$ too little capacity at 1:8), nearly useless and thus leads to near zero collected samples. However, this is different for flow 2 of the optimised implementations

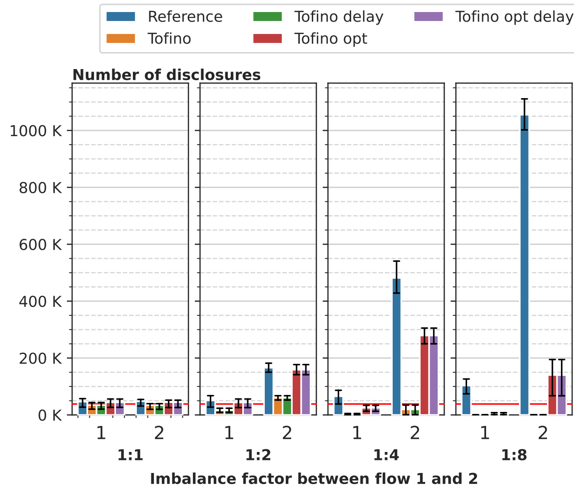


Figure 5.14: Number of samples collected within a minute using the *imbalanced* dataset. The bars represent the median, and the error bars represent $[min, max]$

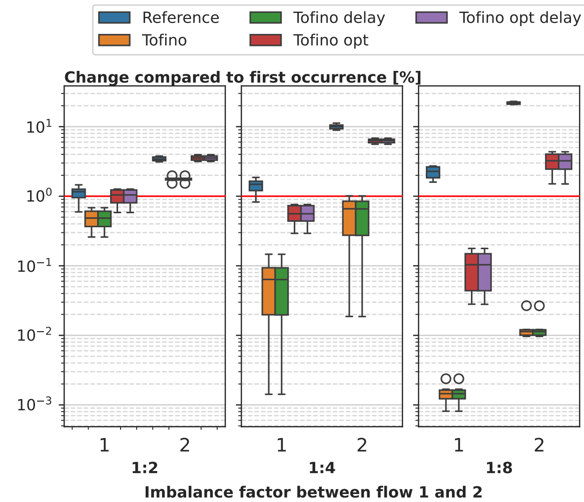


Figure 5.15: Relative change per flow compared to the first occurrence using the *imbalanced* dataset.

(*TofinoOpt* and *delayedTofinoOpt*) with the 7 bytes receipts offering 2x buffer capacity, leading to a buffer that is only 2.5x too small: These implementations manage to collect enough samples for the larger flows, peaking at an imbalance of 1:4, before decreasing due to the too-fast eviction causing the quiet time to impact the receipts.

This shows the behaviour if a flow with a rate below r_{min} is encountered on an otherwise utilized sampling node, showing a weakness of the FIFO RPS Algorithm compared to the RPS Algorithm.

5.3 Performance

To estimate the achievable performance on real hardware and to verify the simulator's results, the *digest* and *annotated FIFO* implementations are run on the Edgecore Networks WEDGE100BF-32Q.

5.3.1 Setup

The available infrastructure of the Networked Systems Group's lab at ETH Zürich was used to obtain results from a real hardware target.

Environment The hardware setup (Figure 5.16) is located in the lab of the NSG and consists of a Edgecore Networks WEDGE100BF-32Q connected to a virtual machine with a dual port NIC. Traffic is sent and received using a Mellanox ConnectX-6 Dx EN[2] with the *mlnx-en-23.10* driver installed. The card is connected using a PCIe Gen 4.0 x16 bus and is directly passed to the virtualised Ubuntu 18.04.6. The NIC is connected to the virtualised two 100 Gbit/s links that use Reed–Solomon error correction. On the switch side, an Edgecore Networks WEDGE100BF-32Q is used that runs Ubuntu 16.04.6 LTS on the BMC and has the Intel® P4 Studio SDE 9.9 installed.

[2]MCX623106AS-CDAT

The two 10 Gbit/s interfaces are implemented as an on-board Intel® Ethernet Controller X552 10 GbE SFP+ dual-port network card using the *ixgbe* driver in the version 4.2.1. Communication from and to the Tofino™ ASIC is facilitated through a PCIe Gen 2 x4 interface, managed by the Intel®'s supplied kernel driver.

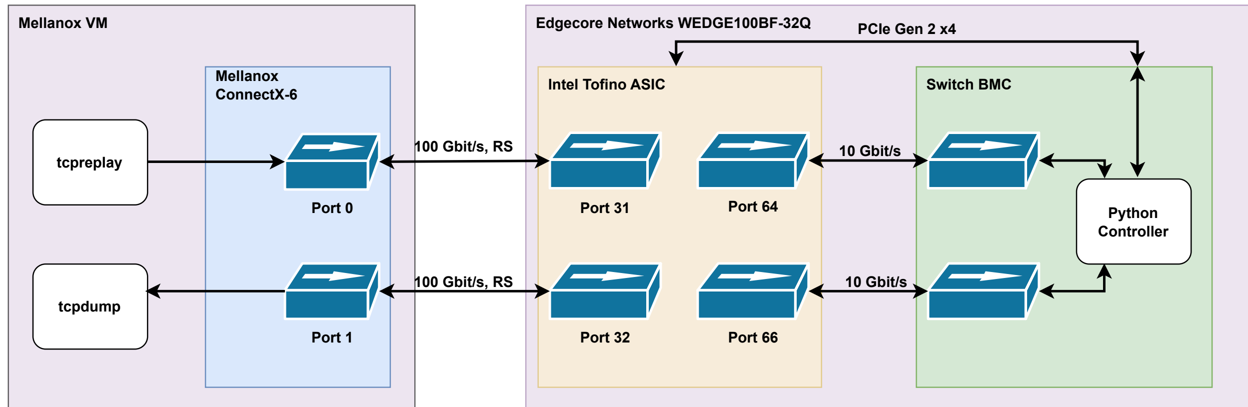


Figure 5.16: Connection overview of the testbed.

Baseline Using *iperf-3.16*, and the Tofino™ configured using an L2 forwarding P4 program, a performance of 90 Gbit/s could be established. Although there are some drawbacks of sending and receiving on the same machine, and the machine being virtualised has drawbacks in performance, these do not affect the measurements performed in this work.

Controller Implementation The controller for the P4 program is implemented in Python and uses the *BF Runtime API* to interact with the switching ASIC.

5.3.2 Methodology

The first combined trace of the *balanced* dataset introduced in Section 5.2.2 is reused. Additionally, the pre-computed expected disclosures obtained in the previous experiments of the *Reference* implementation are used as a reference. The algorithm parametrisation presented in Section 5.2.2 is reused.

A reduced dataset called *balanced_reduced*, consisting of the first 10 million packets of the first combined trace with two flows from the *balanced* dataset, is used additionally for a more detailed analysis of implementation behaviour. The trace lasts 9.2 seconds and has an average packet rate of 1.083 Mpps.

5.3.3 Implementation

The following implementation was chosen to run on the hardware to be compared against the *Reference*:

Digest The digest implementation uses *LearningFilter* to transmit the 12 bytes wide receipts to the controller. The buffering and delayed disclosure of receipts is handled in the controller. The implementation is introduced in Section 4.2.2.

The implementation for sampling on the controller using the Ethernet interface was not evaluated due to early performance issues using the Python network library *scapy* [42], allowing only around 3′000 Kpps traffic to be processable without drops. Furthermore, complications with the Tofino™ and the X552 that could not be resolved on time, led to the omission.

The timestamp-based implementations were not evaluated due to the computational overhead in computing time ranges in the controller, leaving the implementation and evaluation to future work.

5.3.4 Controller-based sampling on the Tofino™

Although the controller-based implementations do not satisfy the design goal of pushing as much complexity to the data plane, the accuracy of the *Digest* implementation is evaluated. We find that only 1.5% of all receipts generated in the data plane are processed by the controller due to overheads in the used controller framework. Hence, the theoretical performance of 141 Mpps, as introduced in Section 4.2.2, is far from reachable using our implementation, handling around 4 Kpps.

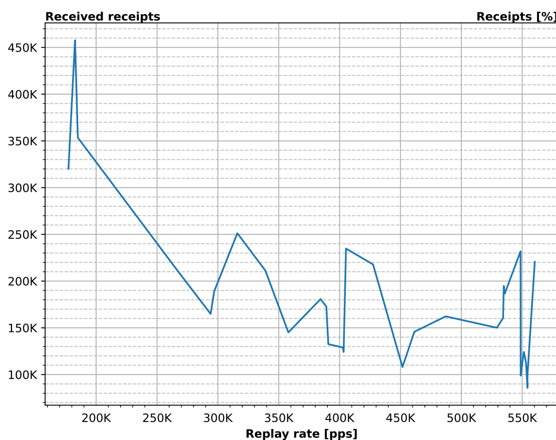


Figure 5.17: Number of collected receipts using *balanced_reduced* whilst replaying at different speeds.

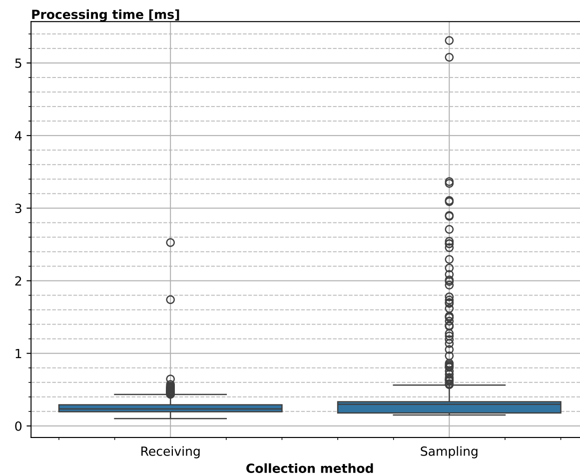


Figure 5.18: Time between two subsequent processing calls. This includes the time to fetch the data and then process it. Results are based on *balanced_reduced*.

Sustaining digest rates As *Digest* relies on the RPS Algorithm, the sampling is identical to *Reference*. Thus, it needs to be shown that all generated receipts in the data plane reach the control plane. Replaying the *balanced_reduced* trace at various speeds, resulting in different packet rates, we count the number of receipts received by the control plane.

Around 1.5% of all receipts generated in the data plane are received by the Python controller for rates above 300 Kpps and around 3.5% for rates below until the testing minimum at 200 Kpps is reached (Figure 5.17). Lowering the sending rate, we found that no more receipts are lost at a rate of 4 Kpps and below. This result is off by a factor of 35′250 from the theoretically achievable 141 Mpps receipts transmitted. The low performance can be explained by the used *APIs* to obtain the receipts. The implementation uses the *BF Runtime API*, which offers a programming language agnostic interface due to the communication with a *gRPC* [43] server to exchange data

and commands. On the one hand, this adds delay, whilst, on the other hand, Python adds an overhead in changing the data representation. The data is parsed from the request and placed in a list of dictionaries offered to the high-level controller. However, it is unclear if a highly-optimized implementation will reach the theoretical rates, as the digest bandwidth is not documented.

Processing delay The *RPS paper* implementation added an extra latency of 1.2 *us* for every packet on average. Our implementation requires an average 79 *us* processing time in the controller, and the performance of the data plane remains unaffected. To measure the performance impact on the controller, the time between two subsequent polls for a digest is measured, including the number of digests contained. To get the overhead added by the RPS Algorithm implemented in Python, the *Receive* latency, consisting of just getting the receipts compared, is compared to the *Sampling* latency, which adds the time to add the receipt to the RPS Algorithm.

Using the *balanced_reduced* trace and a buffer size of 171′000, the average receipt processing time, including the digest acquisition, is 333 *us*. Compared to just receiving the receipt, which takes an average of 254 *us*, the sampling adds an overhead of 79 *us* (Figure 5.18), requiring 31% more time per packet. But this is just the average, delayed disclosures cause delay spikes of up to 5 *ms*.

Chapter 6

Conclusion and Outlook

We have shown that RPS can be run in the data plane by adapting the sampling algorithm from an implementation in a general-purpose language into P4. In the experiments on the *balanced* dataset, we showed that the changes made to the algorithm only affect the sampling performance by around 10%, while maintaining the target accuracy on both datasets on average in all but one traces. When simulating the possible performance on a Tofino™, we found a maximal sampling capacity per pipeline of 2.38 Mpps at a $r_{min} = 200$ Kpps, equivalent to 15.29 Gbit/s at an average size of 800 bytes per packet. This sampling capacity can be sustained whilst utilizing 30% of the overall SRAM available per pipeline. By implementing a hybrid without relying on the integrated switch NIC, we found performance limitations when using Python and the *Bf Runtime API*. Furthermore, we identified the possibility of improving the overall sampling performance in the data plane by increasing the selection rate σ by 50% to 1.5%, achieving a sampling capacity of 3.18 Mpps (20 Gbit/s) per pipeline.

Takeaways Our results indicate that in-data plane sampling is possible with P4-enabled switches on the market is possible. Bringing the advantage of minimizing the additional network bandwidth for sampling and not impacting the forwarding latency by removing the need for a dedicated sampling device. Whilst these benefits are useful for specific situations, we see limited usefulness on the Tofino™ in a pure in-data plane sampling configuration, as utilizing the NIC to the BMC and running the reference implementation leads to $3x$ the performance. An ISP might favour this approach as it allows for more extensive usage of the tight SRAM resources on the device. Furthermore, a monitor entity might also favour this approach as more fine-grained flows can be defined with the available host memory two to three orders of magnitude larger than the Tofino™'s in-data plane memory.

6.1 Future work

We identify multiple opportunities for future work based on this work's observations. These include an optimised RPS controller to achieve the best performance on the Tofino™, allowing for extensive testing, an evaluation of different devices and their storage capabilities and the need for an overall framework for RPS to allow for efficient optimisations.

Optimisation Except for the simulator, this work used Python implementations that proved to be a bottleneck with increasing traffic rates. Hence, we recommend the implementation of a Tofino™-centric sampler using the *C-API* and the Data Plane Developer Kit (DPDK) [17] to guarantee the

best possible performance. Additionally, an upgrade to the Tofino™ 2 is recommended to profit from more stages, larger SRAM and higher PCIe bandwidths. Furthermore, the sampler was the only service running on the switch, and additional effort is needed to integrate it with other services. Lastly, an optimisation based on a fixed number of flows with a per-flow buffer could be implemented and evaluated to prevent the victimisation of small flows.

Extensive testing We used a simulator to understand the sampling performance implications and circumvent the need for a highly optimised implementation. However, with the previously mentioned performance improvements implemented, a real-world experiment can be devised using multiple sampling nodes to show the exact performance. Using traffic generation frameworks like *MoonGen* [44] or *Cisco TRex* [45] to generate loads over 100 Mpps with customisable flows, the impact of the parametrization, such as r_{min}, σ and the operational regime, can be understood in real-time, avoiding the simulation overhead. We tried to use *MoonGen*; however, compatibility issues and the prioritization of the simulator in the available time led to the discontinuation of this path.

Alternative devices With the Tofino™ now being 8 years old, new P4-capable devices have been published. With the trend of increasing SRAM sizes [34], larger receipt buffers can be maintained in the data plane. Furthermore, the implementations are not limited to programmable data planes: *Ribosome* [46], a stateful packet processor, splits headers and processes them on a dedicated server. RPS could be run additionally there, combining advanced stateful processing with RPS.

RPS Framework Due to RPS not being standardized with set rules and having an accessible monitor, optimizations are difficult to perform as they require specific assumptions. We avoided this by comparing the accuracy of the implementations against the reference, but questions such as how the monitor reacts if fewer samples are collected due to buffer limitations or if many late disclosure warnings are reported remain. Both are important as fewer samples reduce the expressiveness of calculated metrics, and an ISP can use many late disclosures to cheat. Hence, the monitor requires additional policies on handling various buffer sizes or the general case, if an ISP cannot provide large enough buffers. Furthermore, RPS requires parameter synchronisation over multiple ISPs to ensure sampling consistency. A protocol that takes the ISPs' traffic rates and available buffer size is needed to facilitate this.

Different hash algorithms and their impact on the overall framework should be evaluated, especially for programmable data planes usage. Finally, to define meaningful aggregates, metadata for each disclosed packet has to be collected when seeing a flow for the first time. Future work can focus on determining if a flow is seen for the first time and how the metadata is collected as additional storage space is required.

6.2 Conclusion

By implementing RPS in P4 for the Tofino™, we showed that sampling in the data plane without additional hardware is possible. Using a simulator, we showed that with the limited memory resources in the data plane as well as with the constrained memory primitives, a performance of up to 56 Gbit/s while utilizing around 30% of the available SRAM can be achieved. We identified the limited memory as the bottleneck of in-data plane sampling and also provided alternatives using a hybrid approach, capable of sampling over 100 Gbit/s independent of the traffic patterns

due to utilizing the host’s main memory. With this hybrid approach, ISPs can prove to their customers that they honour their SLAs with minimal impact on the switches’ feature richness, potentially leading to a more transparent Internet.

Bibliography

[1] Cogent, "Service Level Agreement of Cogent." [Online]. Available: https://www.cogentco.com/files/docs/network/performance/global_sla.pdf

[2] "Internet Control Message Protocol," Internet Engineering Task Force, Request for Comments RFC 792, Sep. 1981, num Pages: 21. [Online]. Available: https://datatracker.ietf.org/doc/rfc792

[3] C. Pappas, K. Argyraki, S. Bechtold, and A. Perrig, "Transparency Instead of Neutrality," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/2834050.2834082

[4] P. Nikolopoulos, C. Pappas, K. Argyraki, and A. Perrig, "Retroactive Packet Sampling for Traffic Receipts," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 19:1–19:39, Mar. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3322205.3311090

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[6] S. Lee, "Reducing complexity of large-scale network configuration management," Ph.D., Carnegie Mellon University, United States – Pennsylvania, 2010, iSBN: 9781124101866 Publication Title: ProQuest Dissertations and Theses. [Online]. Available: https://www.proquest.com/docview/733012977/abstract/95A54DCF02014F18PQ/1

[7] G. Simsek, D. Ergenç, and E. Onur, "Reliable and Distributed Network Monitoring via In-band Network Telemetry," Dec. 2022, arXiv:2212.14876 [cs]. [Online]. Available: http://arxiv.org/abs/2212.14876

[8] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 271–282, Aug. 2000. [Online]. Available: https://dl.acm.org/doi/10.1145/347057.347555

[9] G. Carle, S. Zander, and T. Zseby, "Evaluation of building blocks for pure passive One-way-delay measurements," 2001.

[10] K. Argyraki, P. Maniatis, and A. Singla, "Verifiable Network-Performance Measurements," in *Proceedings of the 6th International COnference*, Nov. 2010, pp. 1–12, arXiv:1005.3148 [cs]. [Online]. Available: http://arxiv.org/abs/1005.3148

[11] X. Zhang, A. Jain, and A. Perrig, “Packet-dropping adversary identification for data plane security,” in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT ’08. New York, NY, USA: Association for Computing Machinery, Dec. 2008, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/1544012.1544036

[12] J. Tyson, “Howstuffworks ”How Internet Infrastructure Works”.” [Online]. Available: https://web.stanford.edu/class/msande91si/www-spr04/readings/week1/Howstuffworks.htm

[13] “The CAIDA UCSD Anonymized Internet Traces - 2018,” Apr. 2018. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[14] M. Rouse, “Optical Carrier,” Jan. 2014. [Online]. Available: https://www.techopedia.com/definition/2735/optical-carrier-oc

[15] “Cisco NetFlow Overview.” [Online]. Available: https://www.cisco.com/c/dam/en/us/td/docs/routers/asr920/configuration/guide/netmgmt/fnf-xe-3e-asr920-book.html

[16] G. Harris and M. Richardson, “PCAP Capture File Format,” Internet Engineering Task Force, Internet Draft draft-gharris-opsawg-pcap-01, Dec. 2020, num Pages: 29. [Online]. Available: https://datatracker.ietf.org/doc/draft-gharris-opsawg-pcap-01

[17] T. L. Foundation, “Home.” [Online]. Available: https://www.dpdk.org/

[18] D. Bahr, “d-bahr/CRCpp,” Mar. 2024, original-date: 2016-05-01T06:24:31Z. [Online]. Available: https://github.com/d-bahr/CRCpp

[19] S. Gueron, “Intel’s New AES Instructions for Enhanced Performance and Security,” in *Fast Software Encryption*, O. Dunkelman, Ed. Berlin, Heidelberg: Springer, 2009, pp. 51–66.

[20] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, “A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research,” *arXiv:2101.10632 [cs]*, Jan. 2021, arXiv: 2101.10632. [Online]. Available: http://arxiv.org/abs/2101.10632

[21] C. Klopfstein, “A benchmark suite to estimate potential performance of sPIN on switches,” Feb. 2021.

[22] T. P. L. Consortium, “P4˜16˜ Language Specification,” May 2023. [Online]. Available: https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.html

[23] “p4lang/behavioral-model,” Jan. 2024, original-date: 2015-01-26T21:43:23Z. [Online]. Available: https://github.com/p4lang/behavioral-model

[24] A. Agrawal and C. Kim, “Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch.” IEEE Computer Society, Aug. 2020, pp. 1–32. [Online]. Available: https://www.computer.org/csdl/proceedings-article/hcs/2020/09220636/1nTub1lNEwU

[25] C. Intel, “Intel® Tofino 2 12.8 Tbps, 20 stage, 4 pipelines - Product Specifications.” [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html

[26] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, “An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends,” *arXiv:2102.00643 [cs]*, Feb. 2021, arXiv: 2102.00643. [Online]. Available: http://arxiv.org/abs/2102.00643

[27] "Intel® Tofino™ Series Programmable Ethernet Switch ASIC," Jun. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html

[28] J. Heinanen and R. Guerin, "A Two Rate Three Color Marker," Internet Engineering Task Force, Request for Comments RFC 2698, Sep. 1999, num Pages: 5. [Online]. Available: https://datatracker.ietf.org/doc/rfc2698

[29] X. Chen, "Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, ser. SPIN '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 8–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3405669.3405819

[30] "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," Internet Engineering Task Force, Request for Comments RFC 826, Nov. 1982, num Pages: 10. [Online]. Available: https://datatracker.ietf.org/doc/rfc826

[31] M. Stigge, H. Plötz, W. Müller, and J. Redlich, "Reversing CRC { Theory and Practice," 2006. [Online]. Available: https://www.semanticscholar.org/paper/Reversing-CRC-%7B-Theory-and-Practice-Stigge-Pl%C3%B6tz/03ccbe5cd0650fa7e69cae2b655b035c7e574685

[32] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler, "Flare: Flexible In-Network Allreduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2021, pp. 1–16, arXiv:2106.15565 [cs]. [Online]. Available: http://arxiv.org/abs/2106.15565

[33] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing," 2022, pp. 1345–1358. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/zeng

[34] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 15–28. [Online]. Available: https://dl.acm.org/doi/10.1145/3098822.3098824

[35] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle, "Demystifying the Performance of XDP BPF," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2019, pp. 208–212. [Online]. Available: https://ieeexplore.ieee.org/document/8806651/references#references

[36] "Cisco IOS NetFlow." [Online]. Available: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[37] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*, pp. 1–499, Jun. 2020, conference Name: IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008). [Online]. Available: https://ieeexplore.ieee.org/document/9120376

[38] "nsg-ethz/p4-utils," Mar. 2024, original-date: 2017-11-13T11:50:15Z. [Online]. Available: https://github.com/nsg-ethz/p4-utils

[39] H. Zolfaghari, D. Rossi, and J. Nurmi, "Reducing Crossbar Costs in the Match-Action Pipeline," in *2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, May 2019, pp. 1–6, iSSN: 2325-5609. [Online]. Available: https://ieeexplore.ieee.org/document/8808105

[40] Cogent, "Network Map of Cogent." [Online]. Available: https://www.cogentco.com/en/network/network-map

[41] H. Stubbe, S. Gallenmüller, M. Simon, E. Hauser, D. Scholz, and G. Carle, "Keeping up to Date with P4Runtime: An Analysis of Data Plane Updates on P4 Switches," in *2023 IFIP Networking Conference (IFIP Networking)*, Jun. 2023, pp. 1–9, iSSN: 1861-2288. [Online]. Available: https://ieeexplore.ieee.org/document/10186439

[42] "Scapy Introduction — Scapy 2.5.0 documentation," Mar. 2024. [Online]. Available: https://scapy.readthedocs.io/en/latest/introduction.html

[43] "gRPC." [Online]. Available: https://grpc.io/

[44] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[45] "TRex," 2024. [Online]. Available: https://trex-tgn.cisco.com/

[46] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A {High-Speed} Stateful Packet Processing Approach for Tbps Programmable Switches," 2023, pp. 1237–1255. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/scazzariello

Appendix A

Average packet size based on CAIDA traces

The *RPS paper* uses 500 bytes as the average packet size on the internet. Investigating this using the CAIDA [13] traces from 2018, we found an average of 800 bytes more likely. We reported the average packet rate using `capinfos` and plotted this average grouped by the collection day, as shown in Figure A.1. All traces have a rate above 800 pps, except for the 21.09.2018, which had a median of the averages at 790 bytes per packet.

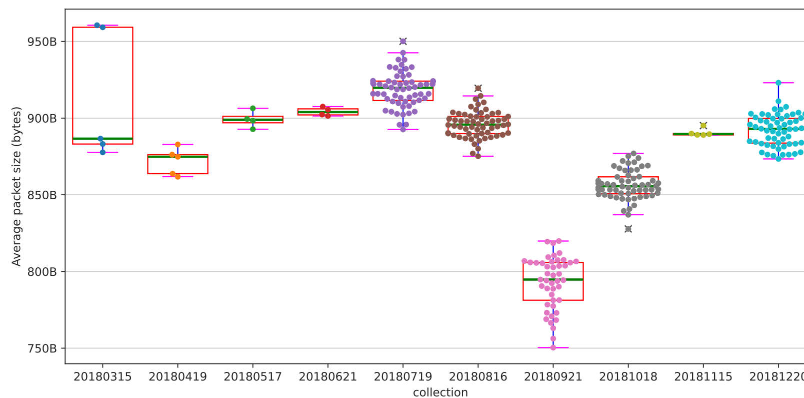


Figure A.1: Overview of the average packet sizes in CAIDA traces based on the collection day.

Appendix B

Trace configuration

The following CAIDA [13] traces were used to create the *balanced* (Table B.1) and *imbalanced* (Table B.2) traces. For the *balanced* traces, an experiment with 4 flows for a given trace out of the six contains the flows $1, 2, 3$ and 4. The *imbalanced* traces consists of a *lower-rate flow* with the *flowid* 1 and a *higher-rate flow* with *flowid* 2. The imbalance factor is the ratio of traces and signifies the approximate rate ratio between the *lower-rate flow* and the *higher-rate flow*. A trace of imbalance 1:4 consists of the trace of the *lower-rate flow* and the combination of the traces of the imbalance factors 1:1, 1:2, 1:3 and 1:4. All traces are extracted from the point *equinix-nyc.dirA* and use the in-file timestamps with microsecond precision. The combiner application merges used combines the traces by timestamp and *flowid* and writes the output file with nanosecond precision. Hence, if timestamps are equal, the *flowid* will be used as the following ordering criterion.

flowid	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6
1	20180719-130000	20180315-130000	20181018-130300	20180419-130300	20180719-130100	20180719-130700
2	20180816-130000	20181115-130400	20180315-130000	20180517-130600	20180517-130500	20180621-130000
3	20180921-130000	20180621-130800	20180621-130800	20180315-132900	20181018-130200	20180419-131400
4	20181018-130000	20180419-130600	20180816-130300	20181115-130100	20180315-130300	20181220-130000
5	20181115-130000	20181220-130200	20180419-131000	20181220-130100	20181115-130400	20181115-130300
6	20181220-130000	20180921-130300	20180719-130400	20180921-130100	20180419-131500	20180816-130400
7	20180517-130300	20180719-130800	20180517-130300	20180719-130800	20180921-130200	20180719-130300
8	20180621-130000	20181018-130400	20180921-130000	20181018-130400	20180816-130100	20181220-130300

Table B.1: Trace configuration for the balanced flows. Every trace is used as a sample trace. The full name, as available on the website, is *equinix-nyc.dirA.< entry >.UTC.anon.pcap*

flowid	ratio	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6
1	-	20180816-130000	20181018-130100	20180921-130200	20180517-130500	20181115-130400	20180719-130100
2	(1:1)	20180719-130000	20180816-130200	20181018-130400	20180719-130500	20180921-130200	20180517-133000
2	(1:2)	20180719-130100	20180816-132600	20181018-130100	20180719-130600	20180921-132800	20180517-132700
2	(1:3)	20180719-130200	20180816-130000	20181018-130000	20180719-130200	20180921-130100	20180517-130300
2	(1:4)	20180719-130300	20180816-132400	20181018-130300	20180719-130000	20180921-130000	20180517-125910
2	(1:5)	20180719-130400	20180816-130300	20181018-131600	20180719-130800	20180921-132700	20180517-130400
2	(1:6)	20180719-130500	20180816-132300	20181018-130200	20180719-130900	20180921-130300	20180517-133300
2	(1:7)	20180719-130600	20180816-130400	20181018-131700	20180719-130100	20180921-132900	20180517-130600
2	(1:8)	20180719-130700	20180816-132500	20181018-131500	20180719-130400	20180921-133000	20180517-130500

Table B.2: Trace configuration for the imbalanced flows. Every trace is used as a sample trace. The full name, as available on the website, is *equinix-nyc.dirA.< entry >.UTC.anon.pcap*

Appendix C

Evaluation of the ideal RPS Algorithm

Although the evaluation of the *Ideal* is not the main goal of this work, we found the obtained relation helpful in understanding the overall sampling performance that could be achieved with an ideal buffer. Hence, the following two implementations that can be run on a generic architecture are compared:

Idea The ideal implementation uses the RPS Algorithm and features an infinitely large buffer, as opposed to the limited buffer size based on the RPS Algorithm parameters.

Reference The reference implementation uses the RPS Algorithm and has a finite buffer based on the RPS parameters.

C.0.1 Evaluation

We provide this baseline as the RPS paper never presented the exact effect and number of disclosures missed due to the reference's buffer size limitation. The reference implementation only discloses around 20% of all possible receipts. Nevertheless, the amount is sufficient to reach the desired accuracy goal.

Stability Increasing the number of similarly scaled flows, the number of collected disclosures remains without large changes and in all cases, the required number of samples are collected (Figure C.1). This expected behaviour is explained by removing receipts from arbitrary positions in the buffer and is a key factor of the RPS Algorithm. The arbitrary removals allow a 'per-flow' buffer to share excess capacity with other flows. Although the large difference of $5-6x$ in collected disclosures between the implementation might suggest that *Reference* would perform badly compared to the *Ideal*. However, if an identical number of samples must be collected with *Reference*, a much larger buffer would be required, missing the goal behind RPS to use as little memory as possible. Furthermore, *Ideal* collects $8x$ more disclosures than the monitor needs within a minute.

Capacity sharing Moving to an imbalanced traffic pattern, the number of disclosures for a lower-rate flow increases as the overall buffer size is increased, as displayed in Figure C.2. This behaviour can again be attributed to the design of the RPS Algorithm, as the larger-rate flow makes the excess space available to a lower-rate flow. Comparing disclosures of the *Reference* from the ratio 1:1 to 1:8, the absolute disclosure count doubles without explicitly allocating buffer

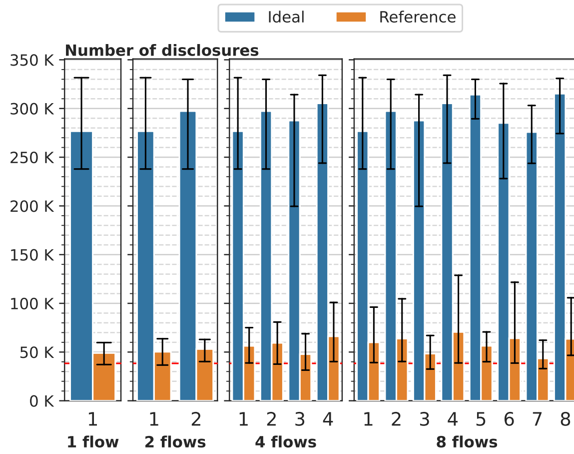


Figure C.1: Absolute number of collected disclosures within 1 minute based on the *balanced* dataset, the error bars represent the *min* and *max*.

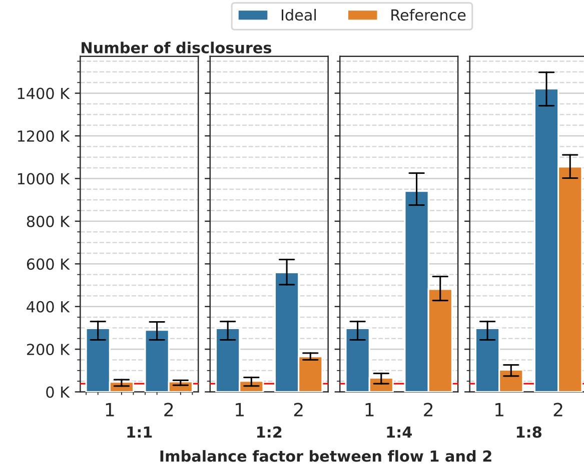


Figure C.2: Absolute number of collected disclosures within 1 minute based on the *imbalanced* dataset, the error bars represent the *min* and *max*.

capacity for the flow, due to the shared buffer. This again demonstrates the strength of the buffer sharing in the case of free capacity.