Seiðr: Dataplane Assisted Flow Classification Using ML

Kyle A. Simpson
University of Glasgow
Glasgow, Scotland, United Kingdom
k.simpson.1@research.gla.ac.uk

Richard Cziva
Lawrence Berkeley National Laboratory
Berkeley, CA, United States
rcziva@lbl.gov

Dimitrios P. Pezaros
University of Glasgow
Glasgow, Scotland, United Kingdom
dimitrios.pezaros@glasgow.ac.uk

***Abstract*—Real-time, high-speed flow classification is fundamental for network operation tasks, including reactive and proactive traffic engineering, anomaly detection and security enhancement. Existing flow classification solutions, however, do not allow operators to classify traffic based on fine-grained, temporal dynamics due to imprecise timing, often rely on sampled data, or only work with low traffic volumes and rates. In this paper, we present Seiðr, a classification solution that: (i) uses precision timing, (ii) has the ability to examine every packet on the network, (iii) classifies very high traffic volumes with high precision. To achieve this, Seiðr exploits the data aggregation and timestamping functionality of programmable dataplanes. As a concrete example, we present how Seiðr can be used together with Machine Learning algorithms (such as CNN, *k*-NN) to provide accurate, real-time and high-speed TCP congestion control classification, separating TCP BBR from its predecessors with over 88–96 % accuracy and F1-score of 0.864–0.965, while only using 15.5 MiB of memory in the dataplane.**

***Index Terms*—Flow Classification, Congestion Control, Dataplane Programming, Machine Learning, P4, CNN, kNN**

I. Introduction

There has been significant research and development on real-time analysis of operational Internet traffic. Accurate flow characterisation (or *classification*) can drive intrusion detection, detecting unusual or illegal patterns of network traffic, or to prioritize traffic for certain customers, to provide path-diversity as well as to mark Quality of Service (QoS) of various users and protocols [1], [2]. However, flow classification solutions today can usually only rely on sampled data provided by routers, such as sFlow, Netflow, or IPFIX, along with imprecise timing (μs and ms-level) [3], [4]. While sampled, low-precision telemetry can be used to classify network traffic based on some flow properties (such as port and protocol numbers) [5], it cannot be used to classify based on fine temporal properties (*e.g.*, identifying bursty flows and senders that can cause microbursts and buffer overflow on the network).

On the other hand, full-software solutions for traffic classification have been proposed by commercial vendors (*e.g.*, Barracuda DPI[1]), the open-source community (*e.g.*, Snort [2], Zeek (formerly Bro) [6]), and the research community, with extensible feature sets and algorithms for classification [7]. Unfortunately, these software solutions designed for commodity hardware do not provide accurate timing of packets, and therefore make certain time-critical events hard or impossible to detect (*e.g.*, microbursts [8] or congestion control properties [7]). Moreover, even the most sophisticated software solutions process packets 10 orders of magnitude slower than current backbone traffic of large operators, making them unusable for large-scale operational analysis [9].

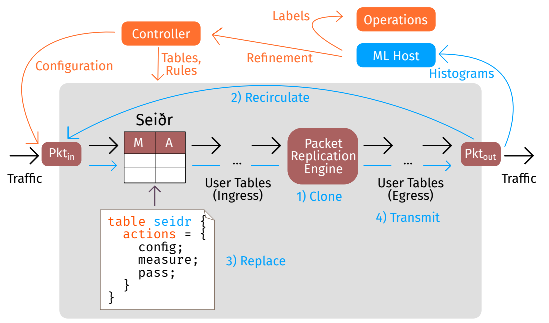
At the same time, programming and fast reconfiguration of network devices is being explored in all types of networks: datacenter and cloud networks, CDNs and WANs. Specifically, with the recent developments of generalized dataplanes (*e.g.*, the *Portable Switch Architecture* [10]), target devices (*e.g.*, Barefoot Tofino and Netronome SmartNICs) along with the high-level programming languages presented for them (*e.g.*, P4 [11]), operators can now express in-network functionality running on their devices, including accurate nanosecond-precision packet timing. However, programming in-network services has its own challenges (*e.g.*, restricted instruction sets, data types and memory), prohibiting the implementation of a fully in-network classification solution.

To solve the aforementioned challenges, we present Seiðr[2], a dataplane assisted flow classification solution. Our design philosophy of Seiðr keeps functionality where it belongs: dataplane devices create accurately timestamped, aggregated data structures for our analysis, and we let a scalable software stack perform ML-based classification on commodity machines. As in-network aggregation reduces the data rate by a factor of ~740, our solution can analyse aggregated data from a total rate of 10 Tbit/s original traffic using a single commodity processing machine.

As a concrete use-case, we look at fine dynamics of TCP congestion control algorithms. Understanding and classifying them is important for network providers as inadequate choices have severe effects on transfer rates, especially in networks with high bandwidth-delay product [12] and in networks where multiple congestion control algorithms are used [13]. By using accurate congestion control diagnostics, operators will be able to infer sender problems (*e.g.*, backlogged or application-limited senders), network inefficiencies (*e.g.*, increased path latency and congestion), as well as receiver issues (*e.g.*, delayed acknowledgements, small receiver windows) and fairness issues between delay-based and loss-based algorithms [13].

[1]https://www.barracuda.com/glossary/deep-packet-inspection

[2]Pronounced "SAY-ther".



```
table seidr {
  actions = {
    config;
    measure;
    pass;
  }
}
```

Figure 1. Seiðr's integration with a PSA-compatible [10] dataplane.

The contributions of this paper are summarized below:

- A flexible dataplane-assisted architecture compatible with the *Portable Switch Architecture* (PSA) [10] that allows data aggregation in the form of histograms with nanosecond-accurate timing (Section II),
- A high-accuracy method for telling apart timer-based (*e.g.*, BBR) and cwnd-based TCP flavours using our system with machine learning algorithms (Section III),
- An extensive evaluation of TCP congestion control classification using our solution (Section IV).

II. Telemetry Creation in the Dataplane

A. Limitations of Programmable Dataplanes

While dataplane programming promises easy reconfiguration of network devices, it poses some challenges. First, network devices support only a limited set of operations and control flows (no loops) without use of platform-specific **`extern`**s, and restrict the user to specific primitive data types, *i.e.*, no floating-point units due to tight hardware constraints. Second, these devices have limited low-latency memory (on the order of a few tens of MBs [14]) and do not provide dynamic memory management. These limitations prohibit complex algorithms from being implemented, but allow certain restricted solutions, such as what is presented in DAPPER [15], where the authors implement a TCP state machine purely in the dataplane.

B. Histogram Generation

Although packet timing information is useful in understanding network and flow behaviour, without volume or packet rate reduction it is prohibitively expensive for hosts to handle each packet. Histogramming acts as the *aggregation step* which makes this class of analysis feasible in high-speed networks. Figure 1 demonstrates how Seiðr, installed as an additional table in any P4 program, records and transmits inter-arrival time histograms. The format for these histogram packets is outlined in fig. 2; we choose to store individual buckets as **`u16`**s, and the number of buckets in any histogram is fixed at compile time. We set this to 100 buckets per histogram. Packets traverse a table which requires 3 actions to be implemented:

1) `config` reads any matched packets as a `seidr_cfg_t` of type `SET_{ MIN, MAX, DST, SRC, LEN }` by using the P4 parser. These update registers 1–5 in table I, dropping any matched packets.

```
header seidr_cfg_t {
    bit<8> function;
    bit<144> payload;
}
```

```
header seidr_t {
    bit<128> src_ip;
    bit<128> dst_ip;
    bit<16> src_port;
    bit<16> dst_port;
    bit<16> eth_type;
    bit<BUCKETS * 16> histo;
}
```

Figure 2. P4 headers for Seiðr configuration and histograms.

2) `measure` calculates the inter-arrival time, update per-flow histograms, and transmits finished histograms to the correct host. We describe its operation in algorithm 1.
3) `pass` ignores packets, and is the default action.

Constructing Seiðr in this manner allows the control plane to install rules to enable/disable runtime reconfiguration as needed, and to monitor as many or as few flows as desired (*i.e.*, using wildcard rules, or exact matching).

The PSA does not have any mechanisms for generating new packets. To circumvent this, any packet which would complete a histogram is tagged for cloning at the end of the ingress pipeline, and recirculation at egress (line 21). This truncated copy returns to Seiðr's table, where we enable the relevant headers, change L2/3 fields, and write out the histogram contents (lines 5–12). The P4 deparser outputs the new protocol stack at egress, and transmits the histogram UDP packet into the network. Event-driven architecture proposals [16] may allow a more natural means of packet generation.

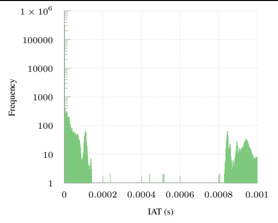
Algorithm 1. Histogram update and transmission.

```
   Data: 5-tuple, P4 metadata, P4 headers, Registers
1  h ← hash(5-tuple);
2  index ← BUCKETS * h;
3  owner ← HistoOwner[h];
4  if metadata.packet_path = RECIRCULATE then
5      headers.tcp.valid ← false;
6      headers.udp.valid ← true;
7      headers.seidr.valid ← true;
8      copy 5-tuple into headers.seidr;
9      rewrite headers.ip, headers.udp using HistoSrc/Dest;
10     headers.seidr.histo ← HistoData[index..];
11     truncate payload;
12     zero out registers: BucketCount, HistoOwner[h],
         HistoData[index..];
13 else if owner = 0 or owner = 5-tuple then
14     HistoOwner[h] ← 5-tuple;
15     iat ← LastTimestamp - metadata.mac_ingress_time;
16     if iat ≥ Min and iat ≤ Max then
17         bucket ← BUCKETS * (iat - Min) / (Max -
             Min);
18         HistoData[index + bucket] ← HistoData[index +
             bucket] + 1;
19         BucketCount[h] ← BucketCount[h] + 1;
20         if BucketCount[h] = Len then
21             mark packet for cloning and recirculation;
```

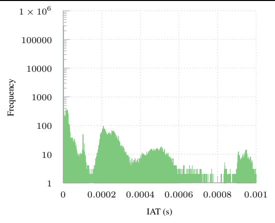
In the event of hash collision (line 13), we ignore packets outside of the tracked flow to ensure that data is accurate. As later processing and classification directly affect what decisions are made by operators or automatically taken by a policy (possibly leading to incorrect flow limits, QoS, *etc.*), avoiding

Table I
REGISTER MAP (DATATYPE, AMOUNT) FOR AN h-BIT HASH.

Min	Max	Length	HistoSrc	HistoDest	BucketCount	LastTimestamp	HistoOwner	HistoData
`u16` 1	`u16` 1	`u16` 1	`u16 + u128` 1	`u16 + u128` 1	`u16` 2^h	`u64` 2^h	`3 * u16 + 2 * u128` 2^h	`BUCKETS * u16` 2^h



(a) TCP Cubic



(b) TCP BBR

Figure 3. Example dataplane histograms showing visible differences in inter-arrival times of selected TCP flavours. Our ML solutions are trained to programmatically identify such differences.

corruption/cross-contamination of operational data is paramount. To gain collision resistance, Robin Hood hashing could be used up to a maximum distance in the table, treating a zeroed owner as empty and an illegal source IP as a tombstone value.

This design allows runtime configuration of all aspects save for the bucket count; at runtime, the only way to increase bucket resolution is to examine a smaller region of IATs. While in theory this could be configured below a maximum compiled into the firmware, the difficulties introduced in classification/data processing make this infeasible. Unless using stream-capable classifiers such as LSTMs [17], changing the input size requires retraining from scratch since new neuron weights must be added and structural properties of the input data change. Increasing the bucket count requires new firmware installation, as many dataplane P4 implementations cannot allocate variable-length stores due to the lack of a dynamic allocator.

As an example of dataplane-generated histograms, fig. 3 shows the distribution of inter-arrival times between two TCP congestion control algorithms. The visible differences are programmatically identified using our ML algorithms.

C. Accurate, Precise and High-Resolution Timestamping

Precise timestamps are critical when detecting temporal properties of flow behaviour, such as microbursts or inferring flow congestion control algorithms. It is especially important in high speed (100 Gbit/s) networks, where there can be as little as 6.7 ns between packets that need to be analysed. With a Linux-based software solution (*e.g.*, reading packets from a link with *tcpdump*), the Linux kernel can only provide microsecond-level accuracy with precision in the order of 100 µs [18]. DPDK improves on this, increasing the accuracy to 100 ns in the best case [19]. However, today's dataplane devices (*e.g.*, Netronome SmartNICs, NetFPGA SUME) allow nanosecond-accurate timestamps to be retrieved from the *Media Access Control* (MAC) modules with a precision of 10 ns [18], a timestamp property Seiðr relies upon.

III. TCP CONGESTION CONTROL CLASSIFICATION

Figure 3 suggests that a notable use-case for this type of measurement is *congestion control algorithm* (CCA) detection. In a TCP connection, each machine is free to choose the CCA it uses to send bytes, and thus how it responds to network congestion signals. This choice is local, and so is invisible to the other machine (and the network). In datacentre networks, operators choose these to ensure optimal behaviour. In a transit network or large WAN however, these hosts (and thus the CCAs in use) are outside the control of network operators, which introduces difficulties when CCA interactions lead to *unfairness*. Consider the recent (and widespread) introduction of *TCP BBR* [12]. *BBR* is a delay/model-based CCA which converges on a fair share of bottleneck bandwidth by reducing its rate if the round-trip time increases, while periodically attempting to increase send rate to account for path/load changes. However, *BBR* traffic can consume 40 % of link capacity when multiplexed with loss-based CCAs, regardless of the number of competing flows [13]. When ensuring fair transit to all flows, this is hardly a desirable outcome; in fact, it's one which may frustrate clients or violate SLAs.

A curious property of *BBR*'s algorithm which sets it apart from other variants is that packet transmission is *timer-based*. `send(packet)`, as defined in the canonical algorithm, asks that on transmission of a packet, the sender should wait for the estimated time that packet would take to reach the recipient. For instance, at an estimated bottleneck bandwidth of 8 Mbit/s, a 1024 kB packet would hold back the next packet in the flow until 976.6 µs had elapsed. When packet sizes remain similar this causes strongly periodic behaviour, while mode switches in the *BBR* algorithm cause these periodic bands to shift up or down accordingly. This effect is stronger than in existing loss- and delay-based algorithms which remain intrinsically tied to the notion of a congestion window (where release of buffered packets follows the receipt of ACK messages). As a result, timing behaviour of past CCAs may be influenced by (the lack of) packet pacing, periodic components might be made noisier by jitter along the return path, or the behaviour of the receiver might add further noise.

This high-level analysis of *BBR* gives us a strong feature to use as the basis for classification: the *inter-arrival times* (IATs) for each packet in a flow. We have two options for processing this for classification: we may use a compressed, fixed-size representation such as histograms to capture the aggregate distribution, or we may attempt to capture structural behaviour by using a variable-length stream of IATs. In many networks, the data and packet rate reduction offered by the former is required to make this possible. Indeed, in-switch aggregation has seen great success in aiding ML for training [20], and direct execution [21]. We make use of the following standard classification algorithms on a fixed-size representation to attempt to single out the CCA in use:

- *k-Nearest Neighbours (k-NN).* A simple and well-understood classifier which assigns labels based on the closest members of the training corpus (*i.e.*, by the L_2 metric). Linear memory cost in amount of training data, and no training cost other than loading all data points, but capable of learning complex decision boundaries on fixed-length input.

Table II
CNN ARCHITECTURE FOR 100-ENTRY HISTOGRAMS.

Layer	Nodes/Filters	Filter Size	Output Dimension
Conv2D	32	(3×1)	$(98 \times 1 \times 32)$
MaxPool	—	(2×1)	$(49 \times 1 \times 32)$
Conv2D	64	(3×1)	$(47 \times 1 \times 64)$
MaxPool	—	(2×1)	$(23 \times 1 \times 64)$
Conv2D	64	(3×1)	$(21 \times 1 \times 64)$
Flatten	—	—	1344
Dense	64	—	64
Dense (Softmax)	$n_{classes}$	—	$n_{classes}$

- *Convolutional Neural Networks (CNNs).* A neural network approach which learns convolution kernels to classify fixed-length data, particularly when recognising spatial features. Memory cost is fixed for a given architecture irrespective of training data, with a high training cost.

When examining k-NN classifiers, we measured accuracy across choices of $k \in [2, 8]$. We found $k = 2$ to be the most effective choice with our input data using the L_2 metric. Our CNN architecture is described in table II, using ReLu activation and 1×1 stride in convolutional layers unless stated otherwise. Training occurred over 5 epochs using the Adam optimiser with categorical cross-entropy as a loss metric, and a batch size of 64 histograms (8 for full sequences due to the smaller data volume). For *BBR vs. Cubic*, the complete model consists of 104 898 32-bit floating-point parameters (409.76 KiB), while the full classification task adds a further 130 parameters (0.51 KiB).

IV. Evaluation

We evaluate the performance of Seiðr from several angles. On classification, we are interested in the accuracy of CCA detection using IAT histograms, the time taken to train a model, and the required time to classify a flow. In the *2-class* problem, we investigate whether it is possible to separate TCP *BBR* from *Cubic* using IAT histograms as the input data, while in the *4-class* problem we extend this to include *Reno* and *Vegas*. We compare our work against Hagos *et al.* [7] in this regard. On deployment, we investigate the bandwidth and memory requirements imposed by Seiðr.

A. Datasets

We examine synthetic flows modelling bulk data transfer at various speeds, generated using iPerf3[3], and processed using custom P4 firmware. For every pairwise interaction between TCP *BBR*, *Cubic*, *Reno*, and *Vegas*, we capture solo and multiplexed dynamics by running each flow for 3 s, with 2 s of overlap (*i.e.*, the second flow begins at $t = 1$ s). We observe that the first flow always completes slow-start before multiplexing begins, and by construction we have several unimpeded captures for every flavour. We do not expect the number/volume of multiplexed flows to substantially alter captured dynamics (*i.e.*, 3 flows at 300 Mbit/s and 2 flows at 200 Mbit/s should both have flows fall to 100 Mbit/s). Flows in one capture are generated using the same target rate in $\{100, 200, \ldots, 1000\}$Mbit/s, each perturbed randomly within $\pm 10\,\%$ uniformly. We also control how this rate limit is applied: *wire-limited* traffic uses `tc` in the Linux kernel to apply rate-limiting, while *application-limited* traffic uses iPerf's builtin mechanisms to control send rate. Application-limited traffic leads to specific behaviour in *BBR* and some other flavours, while wire-limited traffic creates loss events as the rate grows too high. 10 such captures are recorded for each $(CCA_1, CCA_2, speed, limiter)$ tuple, and generated flows are labelled accordingly.

[3]https://iperf.fr/

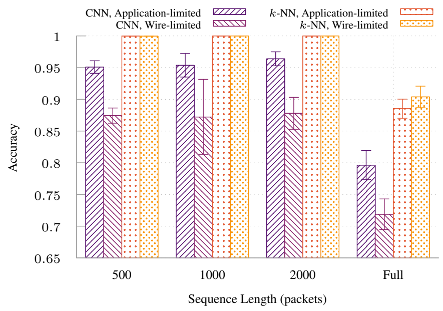


Figure 4. Accuracy of k-NN and CNN classifiers when classifying *BBR* and *Cubic* TCP traffic from IAT histograms, trained over various sequence lengths.

IAT streams are broken down into overlapping sequences of the required length, before being histogrammed into 100 buckets over 0–1 ms. The use of overlapping sequences extends the training and testing sets significantly, while ensuring that larger sequences don't result in a small training corpus. Cross-validation occurs on a per-flow basis rather than per-sequence, *i.e.*, sequences from the same flow must only appear in *either* the test or training set to preserve stringent data hygiene and prevent adjacent sequences from inducing overfitting. All classifier evaluation which follows uses 4-fold cross-validation. The data is comprised of 4994 flows (832 in *2-class*), or 18–31 million sequences (3.2–5.2 million in *2-class*).

B. Experimental Setup

All experiments were executed on Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-96-generic x86_64), using a 4-core Intel Core i7-6700K (clocked at 4.2 GHz) and 32 GB of RAM. CNN training was performed using Nvidia RTX 2080Ti cards (11 GB GDDR6 VRAM). For the dataplane, we used multiple Netronome Agilio CX 2x40GbE SmartNICs using 40GbE connections between source and destination hosts.

C. Classification Performance

In the *2-class* formulation, we observe from fig. 4 that CNN performance increases slightly with the length of the input sequence for classifying application-limited traffic. Our CNN-based detection has a peak F1-score of 0.965 for application-limited traffic, and 0.894 when wire-limited. This increase does not extend towards full-sequence histograms, which are hampered by having 6 orders of magnitude fewer training samples. While very effective, k-NNs come with significant memory cost. By design, the entire dataset must be kept in memory: for length 500, this equates to 1.5 GiB of training data. Naturally, this is undesirable for many network deployments, where easy relocation of inference may be key.

Figure 5 shows in the *4-class* case that we observe a sharp loss in classification accuracy, peaking at $(59.5 \pm 2.0)\,\%$ for

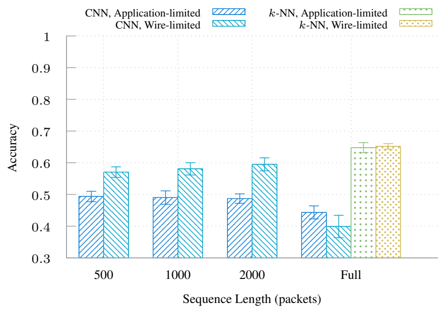
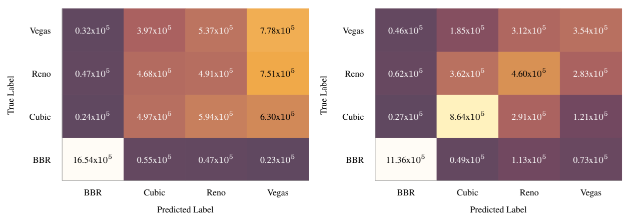


Figure 5. Accuracy of k-NN and CNN classifiers when classifying *BBR*, *Cubic*, *Reno*, and *Vegas* TCP traffic from IAT histograms, trained and tested on various sequence lengths.



(a) *Application-limited.*
$F1_{BBR} = 0.935$, $F1 = 0.486$.

(b) *Wire-limited.*
$F1_{BBR} = 0.893$, $F1 = 0.573$.

Figure 6. Confusion matrices for a CNN on the *4-class* problem, 2000-packet length sequences. Brighter entries along the diagonal indicate correct classifications. BBR remains easy to distinguish regardless of the rate limit mechanism, while Cubic is slightly more distinct for wire-limited traffic.

CNNs and (64.5 ± 1.6) % for k-NN. This suggests that IAT histograms don't generalise as an effective feature for other TCP flavours. Exploratory work with LSTMs on IAT streams confirmed that this persists before aggregation. Likewise, exclusive pairwise training did not lead to an increase in accuracy. However, fig. 6 shows that timing information remains key in separating BBR from its predecessors to a high degree of accuracy, confirming our hypothesis that its *timer*-based (rather than *cwnd*-based) design allows for this detection. If this marker were present between *loss*- and *delay*-based variants, then we'd also see high predictive power over *Vegas* traffic. Breaking down these confusion matrices by rate limit type sheds still more light. In fig. 6a, application-limited data transfers are almost indistinguishable using these metrics (aside from *Vegas*), while fig. 6b reveals that IATs hold some discriminative power for wire-limited Cubic traffic. Note that *4-class* k-NN experiments on all but full sequences required excessive memory and classification time, and so are excluded. While full-sequence k-NNs outperform all examined CNNs on this task (respective peak F1-scores 0.697 vs. 0.486), we observe that these reduce $F1_{BBR}$ from 0.935 to 0.810.

We contrast our work with that of Hagos *et al.* [7], who employ CNNs to predict *cwnd* size for any flow from its stream of bytes-in-flight measurements. On detection of a loss event the *multiplicative decrease* β is measured from estimated *cwnd*s, from which the CCA may be classified. In identifying TCP *BIC*, *Cubic*, and *Reno*, they achieve 95 % accuracy, which outperforms Seiðr on *cwnd*-based CCAs. Yet their approach cannot work for detecting *BBR*. *BBR* is not based upon the notion of a sliding congestion window, so there is no parameter β to infer. Although IAT histograms are suitable for *BBR*

Table III
TRAINING AND INFERENCE COSTS ON OUR TEST SERVER.

Family	Online/Subsequence	$n_{classes}$	Train	Test	Memory
CNN	✓	2	(43 ± 2) min	(49.1 ± 9.2) µs	409.76 KiB
	✓	4	(243 ± 2) min	(50.5 ± 1.7) µs	410.27 KiB
	✗	2	(1.82 ± 0.47) s	(161.3 ± 3.9) µs	409.76 KiB
	✗	4	(7.94 ± 0.50) s	(137.7 ± 1.2) µs	410.27 KiB
k-NN	✓	2	(21.4 ± 1.2) min	(323 ± 69) µs	2.1 GiB
	✓	4	—	—	12.58 GiB
	✗	2	(0.20 ± 0.06) s	(54.0 ± 0.3) µs	332.8 KiB
	✗	4	(2.20 ± 0.04) s	(517.0 ± 5.0) µs	2.0 MiB

detection due to the intrinsic properties of its algorithm, we envision that our approach could be augmented by using a negative *BBR* classification to trigger *cwnd* estimation. Having seen that some predictive power is preserved for *cwnd*-based CCAs, we expect that this will increase the accuracy of a universal classifier. It is important, however, that this step be taken adaptively; this incurs higher resource requirements for bytes-in-flight tracking and for efficient handling of potential return-path asymmetry. Seiðr on its own does not add such overheads or operational complexity, and does not require sight/detection of *cwnd* adjustments.

D. *Training and Inference Costs*

We list typical test and training times for our problem formulations in table III. Training times for k-NN include the time taken to load and process the entire training set, and are incurred *every time* the model is started on a new host. CNNs trained for online analysis (flow subsequences) achieve the lowest per-flow inference times, and are increased during offline analysis due to worse batching and cache behaviour on the smaller data set. While k-NN is effective in many cases, we found it to only be computationally viable when offline (*i.e.*, full-flow histograms), as the entire test data corpus must remain in memory. A single *4-class* cross-validation fold (2000 packets) required 3 days to train and test over the entire dataset, which we deemed infeasible. In contrast, while online CNNs take longer to train, they have a considerably lower memory footprint, the training cost is paid only once, and flows may be classified in real-time with milliseconds of observations.

E. *Switch Resource Usage*

The implementation of Seiðr requires an additional table in the ingress pipeline to update buckets, update configuration, and rewrite packets. Further code space is required to include a configuration packet parser. Shared configuration data (registers 1–5) requires 42 B per switch, while each flow requires 224 B and 248 B to store buckets, counters, previous timestamps, and active 5-tuples on IPv4/v6 networks respectively. On platforms which support hash-table structures, this cost scales linearly with the number of tracked flows. Otherwise, this requires pre-allocation of an entry for every possible hash value (*e.g.*, 14–15.5 MiB for a 16 bit hash). This small memory requirement fits histogram generation to all devices available today.

F. *Quantifying In-Network Data Aggregation*

Figure 7 demonstrates the reduction in data sent from raw mirrored packets, to a stream of measured timestamps/IATs, to Seiðr histograms on an IPv6 network. Timing histograms naturally provide a larger data reduction as the amount of

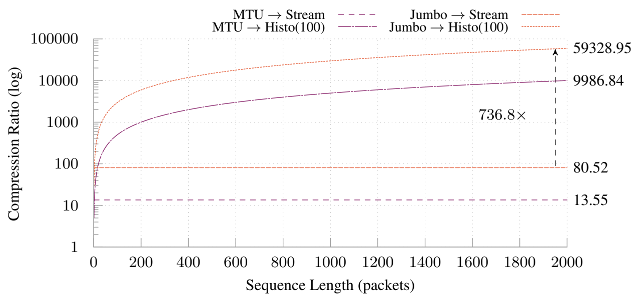


Figure 7. Compression ratio of 100-bucket histograms and timestamp streams from raw packets on an IPv6 network. As sequence length increases, histograms provide more of an advantage in compression rate, being 736.8 × smaller than timestamp streams when analysing 2000-packet sequences.

measured packets increases, while a per-packet IAT/telemetry stream offers no reduction in packet rate. Due to this, 100-bucket histograms cause a greater data reduction than per-packet IATs after just 4 packets in a sequence, and consume 736.8 × less volume for 2000-packet sequences.

To make this concrete, 100 Gbit/s traffic is reduced to 10.01 Mbit/s additional switch traffic for MTU-size packets, and to 1.69 Mbit/s for jumbo frames. IAT streams, by comparison, reduce to 7.38 Gbit/s (resp. 1.24 Gbit/s). For a flow at 100 Mbit/s, only 30 ms is needed to collect enough packets to make a classification. Scaling beyond this, packet processing rates are the bottleneck. As commodity machines and today's stream processors have a reasonable upper bound of ~1M PPS processing capacity [22], Seiðr could scale up to 1 Tbit/s MTU-size packet traffic on one machine, which would correspond to only 333K PPS histogram packets (55.6K PPS if jumbo-size). Reliably scaling to 10 Tbit/s and beyond requires only that we increase the histogram sequence length to ≥ 7000 packets.

V. Conclusion

We have presented Seiðr, a dataplane assisted flow classification solution that can be used to detect fine-grained temporal flow behaviour. We have shown a PSA-compliant way to implement in-network data aggregation in the form of histograms, while using nanosecond-precision timestamping. Our in-network generated histogram datastructure (*e.g.*, on per-flow packet inter-arrival times) has been presented as the input for various ML algorithms, including CNN and *k*-NN. We have shown with our extensive evaluation that Seiðr can successfully tell apart TCP CCAs, in particular, it identifies BBR from its predecessors with over 88–96 % accuracy, while only consuming a maximum 15.5 MiB of dataplane memory. We presented the trade-offs between training and inference times, memory requirements, and accuracy in the context of CNN and *k*-NN classifiers and shown that Seiðr outperforms prior work by increasing classification accuracy on novel TCP CCAs, providing the ability to classify at very high traffic rates (in the order of 10 Tbit/s). Furthermore, we have identified a key temporal property of *BBR* which allows its easy detection among other flows. In the future, we aim to examine the use of Seiðr towards microburst detection and diagnosis [8] and for the identification of *BBR*-like temporal properties of emerging UDP-based congestion-aware protocols, such as *QUIC*.

Acknowledgements

Kyle Simpson was an intern at ESnet, Berkeley Lab. This work was supported in part by the Engineering and Physical Sciences Research Council [grants EP/N509668/1, EP/N033957/1] and European Cooperation in Science and Technology (COST) Action CA15127.

References

[1] L. Bernaille *et al.*, 'Traffic classification on the fly', *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23–26, 2006.
[2] M. Roesch *et al.*, 'Snort: Lightweight intrusion detection for networks.', in *Lisa*, vol. 99, 1999, pp. 229–238.
[3] B. Claise *et al.*, 'Ipfix protocol specification', *Interrnet-draft, work in progress*, 2005.
[4] B. Claise, 'Cisco systems netflow services export version 9', 2004.
[5] D. Rossi and S. Valenti, 'Fine-grained traffic classification with netflow data', in *Proceedings of the 6th international wireless communications and mobile computing conference*, ACM, 2010, pp. 479–483.
[6] V. Paxson *et al.*, 'Bro intrusion detection system', Lawrence Berkeley National Laboratory, Tech. Rep., 2006.
[7] D. H. Hagos *et al.*, 'Towards a robust and scalable TCP flavors prediction model from passive traffic', in *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018*, 2018, pp. 1–11.
[8] X. Chen *et al.*, 'Catching the microburst culprits with snappy', in *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018, Budapest, Hungary, August 24, 2018*, 2018, pp. 22–28.
[9] W. Park and S. Ahn, 'Performance comparison and detection analysis in snort and suricata environment', *Wireless Personal Communications*, vol. 94, no. 2, pp. 241–252, 2017.
[10] The P4.org Architecture Working Group. (Oct. 2019). P4$_{16}$ portable switch architecture (PSA). Working Draft., (visited on 01/11/2019).
[11] P. Bosshart *et al.*, 'P4: programming protocol-independent packet processors', *Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[12] N. Cardwell *et al.*, 'BBR: congestion-based congestion control', *Commun. ACM*, vol. 60, no. 2, pp. 58–66, 2017.
[13] R. Ware *et al.*, 'Modeling bbr's interactions with loss-based congestion control', in *Proceedings of the Internet Measurement Conference, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*, 2019, pp. 137–143.
[14] X. Jin *et al.*, 'Netcache: Balancing key-value stores with fast in-network caching', in *Proceedings of the 26th Symposium on Operating Systems Principles BCP-002*, 2017, pp. 121–136.
[15] M. Ghasemi *et al.*, 'Dapper: Data plane performance diagnosis of TCP', in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, 2017, pp. 61–74.
[16] S. Ibanez *et al.*, 'Event-driven packet processing', in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*, 2019, pp. 133–140.
[17] S. Hochreiter and J. Schmidhuber, 'Long short-term memory', *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[18] R. Kundel *et al.*, 'P4sta: High performance packet timestamping with programmable packet processors', in *IEEE/IFIP Network Operations and Management Symposium NOMS*, *to appear*, 2020.
[19] M. Primorac *et al.*, 'How to measure the killer microsecond', *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 5, pp. 61–66, 2017.
[20] Y. Li *et al.*, 'Accelerating distributed reinforcement learning with in-switch computing', in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 279–291.
[21] Z. Xiong and N. Zilberman, 'Do switches dream of machine learning?: Toward in-network classification', in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*, 2019, pp. 25–33.
[22] A. Gupta *et al.*, 'Sonata: Query-driven streaming network telemetry', in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 357–371.