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

GLOBECOM 2020 - 2020 IEEE Global Communications Conference | 978-1-7281-8298-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/GLOBECOM42002.2020.9348063

***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].

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),

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

[2]Pronounced "SAY-ther".

978-1-7281-8298-8/20/$31.00 ©2020 IEEE

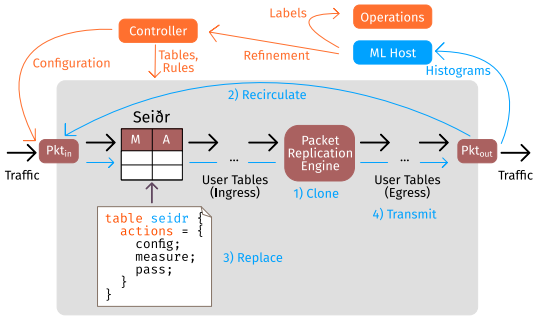


Figure 1. Seiðr's integration with a PSA-compatible [10] dataplane.

- 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.
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).

```
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.

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.

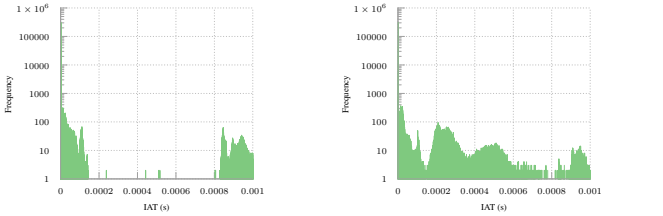
Algorithm 1. Histogram update and transmission.

```
   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 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,

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

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

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.
- *Convolutional Neural Networks (CNNs).* A neural network approach which learns convolution kernels to classify fixed-length data, particularly when recognising

Table II
CNN ARCHITECTURE FOR 100-ENTRY HISTOGRAMS.

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

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,...,1000} Mbit/s, each perturbed randomly within ±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/

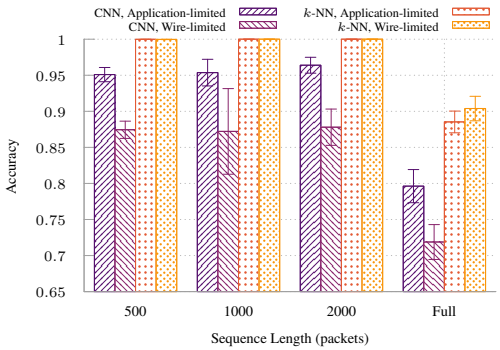


Figure 4. Accuracy of k-NN and CNN classifiers when classifying *BBR* and *Cubic* TCP traffic from IAT histograms, trained over various sequence lengths.

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±2.0) % for 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

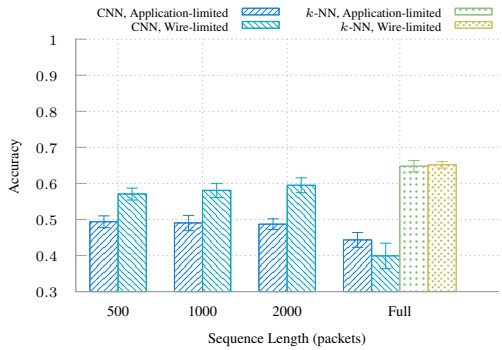
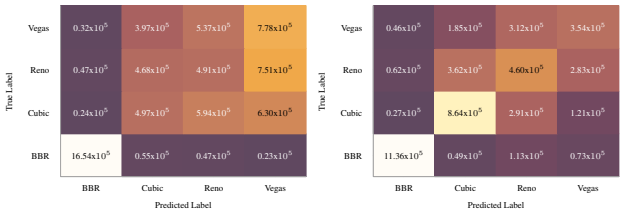


Figure 5. Accuracy of k-NN and CNN classifiers when classifying *BBR*, *Cubic*, *Reno*, and *Vegas* TCP traffic from IAT histograms, trained and tested on various sequence lengths.



(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.

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

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

there is no parameter β to infer. Although IAT histograms are suitable for *BBR* 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,

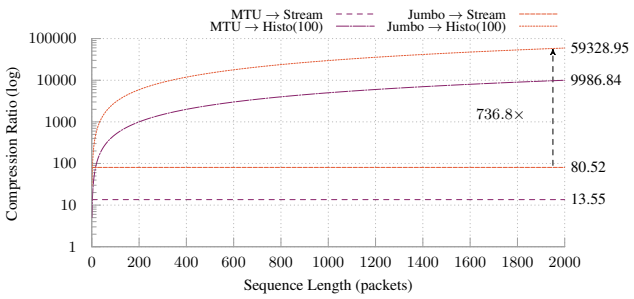


Figure 7. Compression ratio of 100-bucket histograms and timestamp streams from raw packets on an IPv6 network. As sequence length increases, histograms provide more of an advantage in compression rate, being 736.8× smaller than timestamp streams when analysing 2000-packet sequences.

to Seiðr histograms on an IPv6 network. Timing histograms naturally provide a larger data reduction as the amount of 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.

FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications

Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello†, Fernando M. V. Ramos, André Madeira
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{diogo.barradas, nuno.m.santos, ler, fvramos, andre.madeira}@tecnico.ulisboa.pt
†LASIGE, Faculdade de Ciências, Universidade de Lisboa
ssignorello@ciencias.ulisboa.pt

***Abstract*—An emerging trend in network security consists in the adoption of programmable switches for performing various security tasks in large-scale, high-speed networks. However, since existing solutions are tailored to specific tasks, they cannot accommodate a growing variety of ML-based security applications, i.e., security-focused tasks that perform targeted flow classification based on packet size or inter-packet frequency distributions with the help of supervised machine learning algorithms. We present FlowLens, a system that leverages programmable switches to efficiently support multi-purpose ML-based security applications. FlowLens collects features of packet distributions at line speed and classifies flows directly on the switches, enabling network operators to re-purpose this measurement primitive at runtime to serve a different flow classification task. To cope with the resource constraints of programmable switches, FlowLens computes for each flow a memory-efficient representation of relevant features, named "flow marker". Despite its small size, a flow marker contains enough information to perform accurate flow classification. Since flow markers are highly customizable and application-dependent, FlowLens can automatically parameterize the flow marker generation guided by a multi-objective optimization process that can balance their size and accuracy. We evaluated our system in three usage scenarios: covert channel detection, website fingerprinting, and botnet chatter detection. We find that very small markers enable FlowLens to achieve a 150 fold increase in monitoring capacity for covert channel detection with an accuracy drop of only 3% when compared to collecting full packet distributions.**

I. Introduction

Recently, several systems have been proposed for tackling security concerns in modern high-speed networks [90, 33, 49, 82]. By leveraging the capabilities offered by programmable switches, these systems can process packets at line speed directly on the switch hardware, bringing relevant benefits for network security, such as decreased reaction times to attacks, avoidance of network bottlenecks, and decreased costs associated to equivalent centralized server-based infrastructures. So far, the proposed systems target very specific security-driven tasks. These tasks include the ability to mitigate DDoS attacks [90], enforce context-aware security policies [33], obfuscate network topologies [49], filter spoofed traffic [37], or detect data exfiltration through timing covert channels [82].

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24067
www.ndss-symposium.org

However, besides the specific tasks tackled by the previous work, there is currently a lack of support for a new range of security applications that resort to machine learning (ML) to classify flows in real time [92, 27]. This brand of applications has become more relevant as a result of a global trend towards encrypting all Internet traffic [20, 58], which has rendered deep-packet inspection (DPI) increasingly ineffective. As an alternative to DPI, the use of ML-based techniques has proved useful to classify flows with high accuracy for a wide range of scenarios, such as multimedia covert channel detection [7], website fingerprinting [40], botnet traffic identification [53], malware tracking [2], IoT device behavioral analysis [59, 79], or detection of DRM-protected streaming [26, 68, 66].

Most of these ML-based applications rely on supervised machine learning algorithms [7, 40, 68, 53] that need to collect flow features such as packet length and/or inter-packet time frequency distributions. However, both the set of features and ML algorithms used are highly application-dependent. As such, a general service to enable the implementation of ML-based security applications must be versatile enough to accommodate application-specific requirements without impairing its ability to produce accurate classification results. In addition, it must efficiently use the limited switch resources to maximize the number of flows that can be probed, scale to large networks comprising numerous switches, introduce minimal switch downtime caused by upgrades of switch programs, and require low maintenance effort.

We present FlowLens, a system that enables efficient flow classification for multi-purpose ML-based security applications. At the heart of our system lies a set of software components that run on the network switches' data plane and control plane. These components are responsible for collecting compact, but meaningful, features of the flows going past the data plane, and for running the ML-based algorithms responsible for classifying the flows on the control plane in real time. By performing both these tasks on the switches in a fully decentralized fashion, FlowLens does not depend on a centralized service that could introduce bottlenecks for operations in the critical path. To deliver the best performance, these software components must be fine-tuned for each specific ML-based security application. FlowLens includes the mechanisms to generate (and upload to the switches) application-dependent configurations that strike a good balance between classification accuracy and switch resource utilization efficiency. Because these configurations can be automatically generated, the maintenance effort of our system is greatly reduced.

A key challenge in fully offloading ML-based flow analysis onto the switches is tied to the hardware and programming restrictions of modern programmable switches. Ideally, we would like to collect the full packet length and inter-packet arrival time frequency distributions for every flow traveling through the switches. This approach would allow us to collect full per-flow information (on the data plane) which different applications could then process in order to extract relevant features and running specific ML classifiers (on the control plane). However, given that the amount of stateful switch memory is very limited, an information-lossless scheme for collecting flow data would considerably reduce the coverage of our system, i.e., the number of simultaneous flows that could be probed. In alternative, one could employ a lossfull scheme where the amount of dedicated memory allocated per flow is reduced thereby increasing flow coverage. Such a scheme, however, must be such that (1) the collected information does not deteriorate the accuracy of flow classification, (2) it can be implemented with a small set of basic hardware instructions and within few compute cycles as imposed by the switch, and (3) it precludes the need to frequently reprogram the switch as it would cause switch downtime in the order of seconds.

To address this challenge, we make two core technical contributions. First, we devised a new compact representation of packet length and inter-time packet arrival distributions which is small yet provides enough information to perform accurate application-specific traffic classification. We name such representations *flow markers*. We then developed a primitive named Flow Marker Accumulator (FMA) which generates flow markers while depending on simple and efficient operations that can be implemented on modern programmable switches. The FMA consists of a parameterizable data structure deployed on the data plane pipeline such that, for each incoming packet, it performs two simple operations, namely *quantization* and *truncation*, which adjust the granularity of the flow's frequency distribution intervals in bins (quantization), and select the bins considered to be the most relevant features for flow classification (truncation). As shown in Figure 1, the set of resulting bins for each flow constitutes the respective flow marker, which will then be processed by the classification algorithm.

Second, we developed an automatic profiler to find adequate quantization and truncation parameters of the FMA for a given application. Because there is a large space of configurations that present different trade-offs between switch memory savings and flow classification accuracy, manually setting up these parameters for each application would both be cumbersome and render sub-optimal results. Our profiler relies on well-known Bayesian optimization techniques [24] for finding suitable configurations by iteratively testing only a small subset of the possible FMA configurations. It can be tuned to find parameterizations according to different criteria, including (a) the maximization of a user-defined trade-off between space-efficiency and accuracy, (b) the smallest marker able to achieve a classification accuracy above a given threshold, or (c) the marker that maximizes the accuracy given some space constraint.

We have implemented FlowLens on a Barefoot/Intel Tofino programmable switch and evaluated our system in three use case applications: covert channel detection, website fingerprinting, and botnet detection. When comparing the classification scores achieved by FlowLens against those computed over raw packet length distributions, FlowLens offers similar accuracy scores while using significantly less memory, e.g., covert channels can be detected with at most 3% loss in accuracy using only a 20-byte memory footprint per flow. When compared with related methods for capturing compressed packet frequency distributions [22, 55], FlowLens consistently outperforms them in terms of the classification accuracy under similar memory restrictions. FlowLens also achieves considerable bandwidth savings when compared to network telemetry approaches [74] that rely on a server infrastructure responsible for flow analysis.

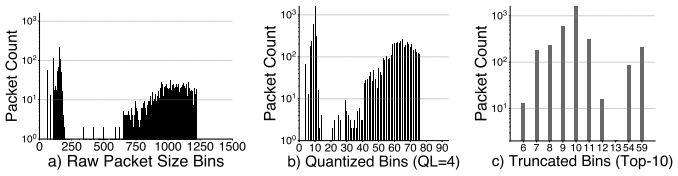


Figure 1. Histograms of packet size distribution for a single flow. A flow consists of a stream of packets identified by the same 5-tuple of TCP/IP header fields, at increasing degrees of compactness: a) the raw packet size distribution; b) the quantized representation, where packet sizes are aggregated into bins of size 2^4 bytes; c) the flow marker generated through truncation of the quantized representation which comprises the most relevant 10 bins for the detection of covert channels mounted through multimedia protocol tunneling.

II. Motivation and Design Goals

This section motivates our work by characterizing a set of emerging ML-based security applications and discussing the technical constraints of modern programmable switches. It then provides an overview of FlowLens's design goals.

A. ML-based Network Security Applications

In recent years, generalized interest has grown in detecting atypical network flows using ML classification algorithms [58]. To deliver accurate flow classification results, these algorithms depend on a range of features that require the collection of the packet length/inter-packet timing frequency distributions. Below, we present three examples of applications in the realm of network security that rely on the analysis of such distributions for performing traffic classification. These examples are chosen to showcase the versatility of FlowLens in accommodating different classification algorithms. We will further use them to validate the classifiers' accuracy when deployed on FlowLens.

Covert channel detection: Capturing packet distributions makes it possible to detect covert channels, thereby providing a valuable asset for cyberforensic investigations. To achieve stealthy data transmissions, advanced covert channel tools tend to obfuscate covert flows such that their high-level features (e.g. packet lengths) resemble those of regular flows [81, 6, 38, 47]. However, recent work [7] has shown that these tools can be defeated or severely weakened due to subtle differences in packet distributions which can be detected by ML techniques.

Website fingerprinting: Privacy-enhancing technologies like OpenSSL or OpenVPN allow users to hide the destination address behind a proxy and the content of website visits from external observers through the use of encryption. However, it may still be possible to identify which sites they access by collecting the flows' packet length distributions [40, 29] and feeding them to ML classification algorithms for website

fingerprinting purposes. This technique may help authorities respond against individuals engaged in illegal activities.

Botnet chatter detection: Botnets [35] can jeopardize the security of multiple organizations, emerging as a highly profitable activity for malicious actors [63]. Unfortunately, due to their decentralized P2P architectures and stealthy communication patterns, botnets have become incredibly resistant to takedown attempts. Nevertheless, state-of-the-art approaches to analyzing botnet traffic are able to identify the presence of bots through the combined analysis of packet lengths and inter-packet timing distributions of network hosts [48, 53]. Being able to employ these techniques can help network administrators prevent and mitigate botnet threats to an organization's network.

B. Design Goals

In this work, our goal is to use programmable switches to collect packet frequency distributions and provide a multi-purpose flow classification platform for implementing a variety of ML-based security applications. By using our solution, a network operator will be able to scan local traffic in near real-time and look for specific flows that match a set of application-specific traffic patterns, such as those presented in Section II-A. In summary, we are driven by the following design goals:

Scalability: We aim at monitoring flows in very large and fast networks (at the Tbps scale), comprised of many switches, while reducing the costs of the network telemetry infrastructure. To this end, we aim to avoid relying either on edge-based solutions, which capture the traffic through middleboxes [61], or solutions that collect packet features on the switches but offload them for further processing and classification on dedicated servers [73, 74]. Reducing the bandwidth consumed with the offloading of telemetry data is also a crucial point as the amount of collected data grows with the increasingly high link speeds and becomes substantial for large scale networks [87].

Accuracy: We aim at collecting a compact representation of packet distributions while retaining enough information about flows to enable high accuracy on classification tasks. Achieving such a representation requires the design of a flow compression scheme that is simple enough to be efficiently implemented by the primitives available in current P4-programmable switches, but which is able to retain meaningful features for classification purposes. Additionally, computing compact representations of packet distributions should not consume the majority of resources in the switch, enabling the system to co-exist with other typical applications, e.g. forwarding, or to be used in tandem with complementary network telemetry solutions.

Availability: Re-purposing our system to different traffic analysis tasks should not involve the deployment of a new P4 program. This is because deploying a new program involves a scheduled downtime during which the switch will be unable to perform its basic functions, causing service disruptions.

C. Constraints of Modern Programmable Switches

To efficiently collect and process packet distributions, we explore the programming capabilities of modern switches, such as Barefoot Tofino [4] and Broadcom Tomahawk II [14]. These switches include two types of processors which operate in two different planes of the network architecture. On the data plane, forwarding ASICs are able to quickly forward and perform simple computations on packets at line-rate, thus enabling the analysis of billions of packets at the Tbps scale. On the control plane, CPUs can be used for general-computing tasks such as controlling the packet forwarding pipeline, or for exchanging data with the ASIC through DMA.

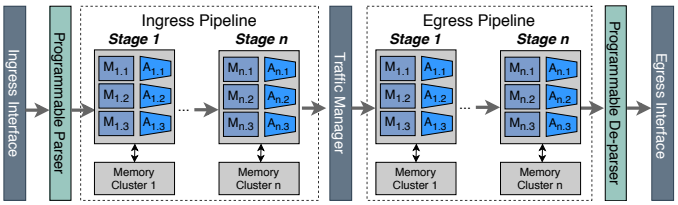


Figure 2. Protocol Independent Switch Architecture (PISA).

Switching ASICs can be programmed in a hardware-independent language, such as P4 [12]. Figure 2 illustrates the architecture of our targeted switching ASIC: the Protocol Independent Switch Architecture (PISA) [18]. Packets arrive at the switch ingress interfaces and, after parsing, are processed by two logical pipelines of *match+action units* (MAUs) arranged in stages. Packet headers along with packet metadata may then *match* (M) a given table, triggering further processing by the *action* (A) unit associated with the matching table's entry. These actions may modify packet header fields and change persistent state (e.g., increment a counter in stateful memory). Tables and other objects defined in a P4 program are instantiated inside MAUs and populated by the control plane at run-time.

Memory constraints: Several constraints in the memory architecture of switching ASICs may restrict the layout of the data structures that can be used by P4 programs. These ASICs are equipped with two high-speed types of memory: (i) TCAM, which is a content addressable memory suited for fast table lookups, such as for longest-prefix matching in routing tables [89], and (ii) SRAM which enables P4 programs to persist state across packets (e.g., using register arrays), and to hold exact-match tables. Unfortunately, switching ASICs contain a small amount of stateful memory (in the order of 100MB SRAM [51]), and only a fraction of the total available SRAM can be used to allocate register arrays. Moreover, accessing all available registers can be a complex task since the registers in one stage cannot be accessed at different stages [19]; this is because the SRAM is uniformly distributed amongst the different stages of the processing pipeline (see Figure 2).

Processing constraints: The P4 programs installed on the switch must also use very simple instructions to process packets. To guarantee line-rate processing, packets must spend a fixed amount of time in each pipeline stage (a few ns [70, 69]) which restricts the number and type of operations allowed within each stage. Multiplications, divisions or floating-point operations, and variable-length loops are not supported. Moreover, each table's action can only perform a restricted set of simpler operations, like additions, bit shifts, and memory accesses that can quickly be performed while the packet is passing through an MAU without stalling the whole pipeline [71].

III. System Overview

This section describes FlowLens, a system for efficient flow classification that achieves the aforementioned goals. Figure 3 shows the architecture of FlowLens, illustrating how it can

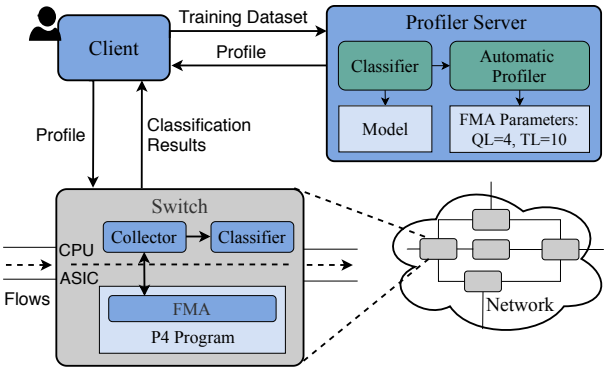


Figure 3. FlowLens architecture and components.

be used to monitor traffic on a single switch of a high-speed network. In general, it can be deployed across multiple other switches in the network at the system operator's discretion. FlowLens consists of the following components: a *P4 program* and two software components (*collector* and *classifier*) running on the switch, a standalone *profiler* server, and a software *client* that provides an interface to the system operator.

Taken together, the components running on the switch are responsible for analyzing traffic and classifying flows as per ML-based security application. The P4 program runs on the data plane, and implements a tailor-made data structure named *Flow Marker Accumulator* (FMA). The FMA is used to collect concise encodings of the packet length or inter-packet timing distributions of flows named *flow markers*. Essentially, the FMA implements the necessary knobs for fine-tuning the memory savings and the classification accuracy of the system. It can accommodate the restrictions of the switch and generate flow markers at different compression levels by carefully adjusting a set of configuration parameters that control (i) the size of the memory footprint allocated per-flow marker, and (ii) the loss of information in the flow markers due to compression.

Running on the local CPU of the switch, two additional FlowLens software components implement several control plane functions. The *collector* is responsible for loading the P4 program, configuring the FMA data structures in the forwarding pipeline, initiating the flow collection process, and collecting the resulting flow markers. The *classifier* runs the ML algorithms responsible for classifying flows based on the collected markers. FlowLens can generally employ any ML algorithm that can reason over flow markers, and whose memory and compute requirements can be accommodated on the switch control plane. After the classification step, results can be downloaded by the client and displayed to the system operator.

Since the classifier and the FMA configuration parameters depend on the application domain, FlowLens uses a standalone *profiler* to pre-configure the classifier models and FMA parameters (i.e., the application *profile*) onto the switch. The system workflow involves two phases: profiling and flow classification.

1. Profiling: FlowLens needs to be pre-configured by the system operator for specific applications. This operation involves the profiler server, which can automatically create profiles by using an application-specific classifier and a training set containing labeled flow samples provided by the system operator. To this end, the profiler runs an optimization process that explores the classification performance of different FMA quantization and truncation values, generating a configuration according to a given user-defined criterion (see Section V).

2. Flow classification: As soon as the P4 program has been loaded into the switch (which happens only once when bootstrapping FlowLens), the collector takes the profile computed in phase 1 for a specific application, e.g., website fingerprinting, and configures the FMA accordingly. Afterward, the switch can start to process packets and compute flow markers. The collector then fetches the resulting flow markers from the data plane and the classifier processes them based on the loaded model. The classifier results can be retrieved by the system operator, who can then take targeted actions about particular flows such as dropping flagged flows or scheduling further logging operations. The system operator can later reconfigure the system for other ML-based application profiles without the need to re-deploy the P4 program.

In the following sections, we present the relevant design details of FlowLens, namely the FMA data structure (Section IV) and the automatic profiling scheme aimed at choosing markers with good accuracy/memory saving trade-offs (Section V).

IV. Flow Marker Accumulator

The Flow Marker Accumulator (FMA) is the data structure responsible for computing flow markers on the switch data plane. Next, we present its internal operations by describing the FMA design for capturing markers of packet length distributions, and presenting the main changes when computing inter-packet arrival timing distributions. Then, we describe how the FMA is used by the control plane, and discuss alternative FMA setups.

A. Collecting Packet Length Distributions

To generate flow markers, the FMA provides two basic operators that can be implemented efficiently and be used to obtain a space-efficient encoding of a packet length distribution: *quantization* and *truncation*. Quantization consists of counting packet lengths in coarse bins that represent ranges of contiguous packet lengths. Truncation further trims the number of bins that need to be reserved for a certain classification task. These operators allow us to selectively collect the bin values which, in many cases, correspond to the most relevant features employed by the ML engine to yield accurate classifications [84, 7].

To perform these operations, the FMA is composed of several data structures shown in Figure 4. They consist of two match+action tables and one register array: the *flow table*, the *truncation table*, and the *register grid*, respectively. The register grid is a matrix of memory registers. Each line is used to store a flow marker. The index of each line (flow offset) is used to address the flow marker. Internally, a flow marker consists of a number of cells in a register (the grid's columns). These cells play the role of bins for storing samples of the flow's packet length frequency distribution. The flow table maps the monitored flows against the respective flow markers in the register grid. The truncation table identifies the bin that must be incremented for every incoming packet.

Next, we describe in detail the procedure for updating the flow marker for a given incoming packet. Consider the example shown in Figure 4, where an input packet arrives in the switch

from source IP 162.2.13.42, source port 41065, with length 1024 bytes. The FMA performs the following four operations:

1. Lookup: First, the FMA's flow table matches the incoming packet with the corresponding flow ID, which is a 5-tuple of header fields $\langle IP_{src}, Port_{src}, IP_{dst}, Port_{dst}, Proto \rangle$ that is used as lookup key to return its associated flow offset. To be efficiently performed, we leverage the match+action units of the switch to accommodate specific rules for flow table indexing. Each rule in the flow table assigns a unique flow offset to each flow ID. For instance, in the running example, the input packet is matched against the rule $\langle 162.2.13.42, 41065, 146.3.18.71, 80, 6 \rangle \rightarrow 0$. (Section IV-C describes how the flow table is populated.)

2. Quantization: The flow offset determined above locates the packet's flow marker in the register grid. Next, the FMA must increment the correct bin in the flow marker which is a function of the packet length. The first step to determine the right bin involves quantization, which aggregates, and so counts, a range of contiguous packet lengths into the same bin. To avoid complex instructions unsupported by the switch hardware (e.g., multiplications), the bin indexed by a certain packet length PL is computed by $bin(QL, PL) = length(PL) >> QL$, where QL denotes the *quantization level* and $0 \leq QL < log_2(PL_{max})$. For efficient lookup of a packet length's bin, FMA uses power-of-two bin sizes; this allows for computing the packet bin by right-shifting the packet length value by QL number of bits. In the shown example, applying $QL = 4$ to the packet length (1024 bytes) yields quantization bin #64.

3. Truncation: Based on the obtained quantization bin, truncation leverages an auxiliary data structure – the truncation table – which contains match+action rules exclusively for the bins that should be accounted for in the flow marker. Each rule is keyed by the quantized bin length, i.e., $bin(QL, PL)$, and indexes the flow marker's bin (bin offset) where the packet length frequency must be recorded. If no such rule exists for a given quantized bin, the packet is not counted. In this example, the current packet is considered because a rule exists for the packet's quantized bin (#64). In contrast, packets whose quantized bin values fall, e.g., within the bin range 52-63, will not be accounted for. This strategy allows for selectively filtering the most meaningful bins for flow classification.

4. Increment: Lastly, by combining the flow offset and the bin offset, the register grid can be indexed and the correct bin incremented. In the running example, this entails incrementing bin 2 of the flow marker pointed to by the flow offset 0. These steps are repeated for every incoming packet.

B. Collecting Packet Timing Distributions

Gathering inter-packet timing distributions requires only minor modifications to the FMA design. This task does not affect the main FMA data structures and operators, but it requires additional resources in the switch processing pipeline.

To compute the arrival time difference between two consecutive packets of the same flow, the FMA stores the timestamp of the last packet seen per each flow. That information is available to the P4 program through device intrinsic metadata. With exception to the first packet of a flow, FMA computes the difference between the current packet timestamp and the last timestamp observed for that particular flow. That value can then be processed by the same quantization and truncation operators described for packet lengths, which produce the corresponding bin to be updated in the register grid.

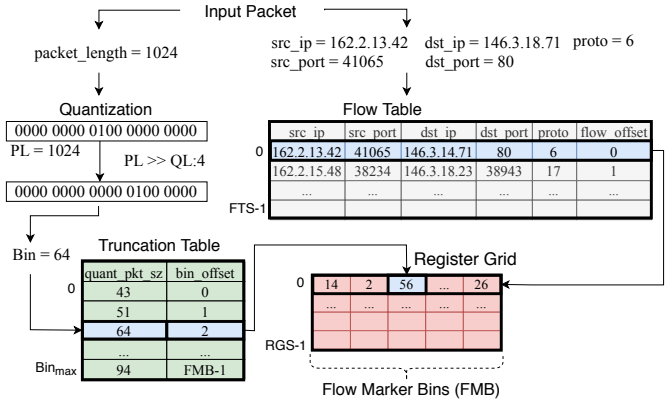


Figure 4. FMA internals: Flow Table, Truncation Table, and Register Grid.

C. Usage of the FMA by the Control Plane

To monitor flows on a switch, the FMA must be coordinated by the control plane's collector software. The collector predefines the FMA's quantization and truncation parameters, and determines: i) which amongst all flows traversing the switch at a given time will be monitored by the FMA, and ii) for how long the packets of the monitored flows will be considered in the respective flow markers. Next, we present the default FMA operational settings and then describe alternative customization policies that can be enabled by the FlowLens operator.

Default FMA measurement operations: By default, the control plane sets up the FMA to i) compute flow markers for all the flows on a first-come, first-served (FCFS) basis until they exhaust the register grid capacity, and ii) measure the flows' respective packets for a predefined time interval that we refer to as *collection window*. The control plane starts by clearing all registers of the FMA's register grid, and setting a timer for the duration of the collection window. Then, for every incoming packet for which a rule does not exist in the FMA's flow table (i.e., the packet belongs to a new flow), the control plane automatically installs one of two possible rules for that flow. If there is free space in the register grid, then it will install a new rule that points to a free flow marker, allowing the flow to be monitored by the FMA. Otherwise, if the register grid is full, the control plane will install a single wildcard *ignore rule* (featuring a reserved offset value) instructing the FMA to ignore that flow and all subsequent flows until the end of the collection window. When the timer expires, the control plane reads in batch all the computed flow markers. To prevent concurrent updates while reading, the control plane deletes the flow rules prior to reading the registers. Once the registers have been read, a new collection window round can then ensue. A side effect of this design is that FlowLens may skip the packets of a new flow until the flow's respective rule is installed in the switch. However, while this is a limitation of our system, such loss does not impair FlowLens's ability to trace most typical flows as they tend to last longer than a few milliseconds.

Discretionary flow monitoring: Prior to the enforcement of the FCFS flow marker allocation strategy, it is possible to filter which traffic should be monitored and therefore limit the amount

of flows to inspect. For instance, ML-based security applications are frequently focused on traffic that can be identified by common target ports (e.g., HTTP). In other cases, the FlowLens operator may be interested in monitoring flows based on IP or network address ranges. Such a discretionary flow filtering stage can be implemented on the control plane by installing ignore rules for all uninteresting flows based on allow / deny policies provided by the FlowLens operator. Ignore rules can be defined to target a single flow or to perform wildcard matching based on IP and port ranges, specific protocol fields, etc.

Fine-tuning of the collection window: Flow markers should not linger for an arbitrarily large amount of time inside FMA's data structures, as this would prevent the flow marker's memory from being used for other flows. To increase the number of flows that can be monitored at any given time, the collection window can be configured, based on three setups: i) the definition of a fixed window duration (explained above); ii) the use of a specific flag set in FMA data structures that enables the control plane to check for flow termination through a polling procedure; iii) a hybrid of both approaches. While option i) is arguably the simplest alternative, it may lead to memory waste since short-lived flows can occupy the FMA's data structures longer than necessary. In contrast, a polling approach allows FlowLens to pinpoint flows that will receive no new packets (e.g., after detecting FIN packets in the data plane pipeline which signals the termination of a TCP connection), but it may indefinitely keep flows which termination is not explicit (e.g., UDP-based multimedia flows). Thus, a hybrid collection approach allows us to proactively read and reset the FMA data structures in use by terminated flows while preventing long-lived flows to be monitored indefinitely. In other words, this approach fully refreshes the register grid every time the collection window is over and partially updates it whenever a particular row is in condition to be evicted.

Flow marker eviction: A high rate of new flows may saturate the capacity of the FMA data structures and prevent storing flow markers for all flows crossing the switch. In this case, as explained above, the FMA's default strategy is to not track new flows as long as existing flows are still being tracked. As an alternative behavior, it is possible to evict flow markers from the FMA according to an LRU policy. In this case, the control plane keeps track of the oldest flow markers stored by the FMA, and replaces them as new incoming flows cross the switch. The most suitable policy will greatly depend on the expected workload and topology of the network.

D. Distributed and Orchestrated FMA Operation

So far, we explained several design decisions of FlowLens when considering its operation to be contained within a single switch. In this section, we describe how FlowLens can benefit from a deployment in multiple vantage points.

Scaling the number of measured flows: Although the number of flows whose state can be kept by a single switch is limited, it is possible to take advantage of multiple vantage points in the network for monitoring a larger amount of flows. This is akin to the operation of other measurement frameworks [43, 31] and may be accomplished, for instance, by splitting packets coming from different IP address spaces between existing switches in the organization's network infrastructure.

Increasing collection coverage: In the case that our system operates with a maximum collection window (see Section IV-C), reading and resetting FMA's data structures requires a non-negligible amount of time (a few seconds) [36]. This may prevent FlowLens from collecting flow information while these operations take place. To ensure visibility over the network traffic crossing an organization, FlowLens can be deployed in a cascade fashion across an additional switch to intertwine the collection windows of the different switches.

Increasing application coverage: The design of FlowLens is tailored for enabling a single profile to be loaded into a given FMA at any given time. However, a coordinated operation of FlowLens across several switches can provide support to multiple ML-based security applications. For instance, when deployed across multiple switches, one FMA instance may be dedicated to the detection of covert channels, while other is dedicated to the identification of botnet behavior.

V. Automatic Profiling

The flow markers generated by FMA depend on the parameters, i.e., the quantization level and the truncation table, dictated by an application-specific profile which determines how efficiently the switch SRAM will be used and how accurate the flow classification will be. In general, finding the parameters that offer an optimal trade-off would require an exhaustive search of the parameter space. Unfortunately, this is a cumbersome task that requires non-trivial computational resources and time, e.g., automatically exploring the full space of configurations for the botnet detection task (Section VII-F) took one day.

To search on the parameter space for a configuration that offers a good trade-off between flow marker size and classification accuracy, the profiler implements optimization techniques that, albeit may fail to yield the optimal result, usually find near-optimal solutions quickly. Next, we describe the optimization criteria and algorithm employed in FlowLens. Note that FlowLens is not tightly coupled to a specific implementation of the profiler and nothing prevents the use of alternate optimization techniques [72, 9]. Investigating further optimization approaches is outside the scope of this paper.

A. Optimization Criteria

We expect that a FlowLens's operator will want to find a suitable FMA parameterization for any given ML-based security application. Because there is a space/accuracy trade-off in the FMA configuration, we are faced with a multi-criteria optimization problem that does not have a single optimal solution but, instead, has a number of Pareto optimal solutions [57]. The current version of the FlowLens's profiler can approximate three different pre-set points in the Pareto frontier, that can be selected by the system operator:

1. Smaller marker for target accuracy: In this mode, the system operator specifies a target accuracy value to be attained, and the profiler automatically chooses the quantization and truncation parameters that yield the smallest marker that is able to offer the target accuracy. Note that the profiler will not return a configuration if the accuracy set by the user cannot be achieved for the particular dataset under analysis.

2. Best accuracy given a size constraint: Here, the system operator specifies the maximum size for the flow marker and the system automatically picks the quantization and truncation parameters that maximize the classification accuracy, among the configurations explored, without exceeding the target marker size. This constraint also allows us to reduce the search space, since the marker size generated by a set of quantization and truncation parameters is known beforehand.

3. Size vs. accuracy trade-off: Lastly, the profiler can work in a fully automated fashion. In this case, the profiler attempts to maximize an accuracy vs marker size trade-off that is expressed by the following reward function: $\textit{reward} = \alpha \cdot \textit{accuracy} + (1-\alpha)\frac{1}{\textit{marker size}}$. A smaller α attributes less importance to the accuracy in favor of compactness, and vice-versa. Our prototype uses $\alpha = 0.5$, but the system operator can define the value of α as well as a different reward policy of its choosing altogether.

B. Optimization Algorithm

The search space is the product of the different quantization and truncation configurations; on its own, the number of configurations that result from truncation is combinatorial with the number of available bins. To guide the profiler's search, we use an optimization algorithm that consists of two phases. In the first phase, called *search space reduction*, we use domain knowledge to narrow the search, by excluding configurations that are unlikely to offer acceptable results. In the second phase, we resort to *Bayesian optimization* to find a suitable flow marker. We detail these two steps in the next paragraphs.

1. Search space reduction: To reduce the search space, we discretize the domain of quantization parameters, e.g., aggregate bins in powers of two. Then, we leverage a pre-training step for narrowing the truncation space: we first generate a coarser representation of the packet distributions of each sample in the training data according to a given quantization parameter, then we use a classifier to build a model based on these representations. We leverage the fact that most classifiers can output information regarding the top-N most relevant bins for accurate classification. Thus, for each quantization, we constrain the exploration to points that include an increasing number of features from the top-50 (i.e., the configuration that includes only the top-10 bins, the top-20 bins, etc). When the classifier is unable to output the top-N features, we fall back to a simpler strategy to narrow the search space: we sample the input space, exclude bins that have not been observed in the sampled points, and feed the remaining ones to the Bayesian optimizer.

2. Bayesian optimization: To reduce the manual labor required to explore a large space of configurations, we rely on Bayesian optimization, which is a well-known method for optimizing black-box functions and for finding near-optimal solutions with few function evaluations [24]. We optimize the combination of quantization and truncation parameters using the Python Hyperopt [10] software package. In each iteration, the profiler selects a parameterization, trains a classifier using flow markers accordingly generated, and records the classifier accuracy alongside the size of produced flow markers. The next parameterization to sample is selected by the optimizer which we run for a fixed number of iterations. The whole process took us a few hours to complete.

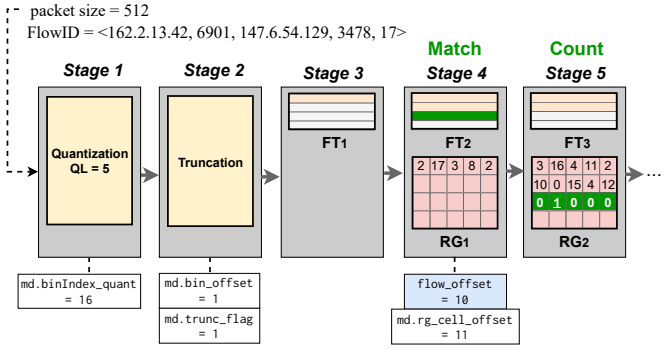


Figure 5. FMA implementation on Tofino switch. Each partition of the flow table (FT_m) records `flow ids` (marked in orange) and their corresponding flow markers are recorded in the register grid partition of the upcoming stage (RG_m). This packet is matched in FT_2 and the respective flow marker updated in RG_2. The white boxes in the bottom indicate the metadata values computed in each stage; the value in the blue box is loaded by the control plane.

VI. Implementation

We built a prototype of the FlowLens system. Excepting the FMA, which was written in P4, we implemented all other components in Python. The classification engine of the profiler server uses Python's `scikit-learn` [62] and the Weka [28] libraries. We implemented FlowLens's FMA [8] for a Barefoot Tofino ASIC [4] using about 500 lines of $P4_{16}$ code, which was compiled with the P4 Studio Software Development Environment (SDE) [5]. While the FMA's design presented in Section IV-A is generically compatible with PISA architecture, its implementation required careful reasoning due to the specific intricacies of currently available switching hardware.

To implement the FMA code for a Tofino switch, we need to fit the FMA's data structures and operations into the specific pipeline and compute capabilities of the switch. To implement flow marker updates, it would be desirable to compute the flow offset and the bin offset of the target flow marker (see Figure 4) in a single pipeline stage to be able to use all the memory in upcoming stages to store flow markers. However, this cannot be achieved on our target hardware due to three major data dependencies: i) matching (i.e., indexing) the truncation table depends on the quantized packet length, but quantization and truncation are too complex to be realized together in a single stage; ii) indexing a flow marker's bin requires the result of truncation, but the truncation table and the flow table cannot be matched in the same physical stage; iii) matching the flow table and updating the respective flow marker are also too complex to perform in a single stage. Moreover, it is not possible to access all the switch memory from a single stage.

Laying out the FMA in hardware: To accommodate for the above requirements, we split the functioning of FMA across different stages, as depicted in Figure 5. To resolve dependencies i) and ii), we reserve the first and second stages of the pipeline to perform quantization and truncation. Then, we partition the flow table and register grid along the remaining stages to use up all the per-stage stateful registers across the processing pipeline. To overcome the inability to calculate the bin offset and increment the corresponding register cell in the same stage – dependency iii) – the flow table partitions and register grid partitions are placed in contiguous stages. Each flow table partition is responsible for managing flow markers in its corresponding register grid partition.

Figure 5 depicts in detail the operation of FMA when a packet for a new flow arrives. Notation FT_m and RG_m denote partition m of the flow table and register grid, respectively. Assume that the collector has installed a rule for `flow id` ⟨162.2.13.42, 6901, 147.6.54.129, 3478, 17⟩ in FT_2, and that the first incoming packet for this flow has a size of 512B. In stage 1, action `quantization_act` is triggered, quantizing the packet length using QL=5 and setting the resulting quantized packet length (`md.binIndex_quant`) to 16. The object `md` stores the metadata carried over across the pipeline stages.

In the second stage, the truncation table matches against the quantized packet length (refer to Figure 4) and triggers the `truncation_act` action, which returns the bin offset `md.bin_offset` within the flow marker and sets a truncation flag `md.trunc_flag` in order to inform the downstream stages that this packet's corresponding flow marker should be incremented. In case no match exists in the truncation table, the truncation flag is not set, and the packet is not accounted for.

Next, since the flow matching rule is not installed in FT_1, the packet is not matched until it reaches stage 4, where FT_2 is located. Upon matching the `flow id` and verifying that `md.trunc_flag` is set, the `set_flow_data_act2` action is triggered. This action computes `md.rg_cell_offset` by adding `md.bin_offset` and `flow_offset` loaded by the control plane into FT_2. The resulting value is used to index a cell in RG_2 which is then incremented. To index the correct partition of the register grid, we use control flow logic to test which flow table partition was matched, triggering the respective `reg_grid_act2` action that updates the flow marker on the corresponding register grid partition in the next pipeline stage.

Optimizing per-packet computations: To reduce the complexity of the P4 program, we leverage the capabilities offered by the control plane to offload the computation of complex operations from the data plane. This results in a program sampled in Listing 1 which implements four simple actions that can be performed within single stages of the pipeline. Specifically, we offload two operations into the control plane:

a) Computing a flow offset within the register grid: The index of the register grid where a flow marker is located can be easily calculated in the control plane. Since the FMA parameterization is known prior to the loading of the P4 program on the switch, the control plane can compute the number of bins used by a flow marker in a given configuration. Thus, when a new flow is matched, the collector installs a rule where the flow offset is given by the number of flow rules installed in a particular flow table partition times the number of bins composing a marker. Upon matching, the flow offset is passed as an argument to action `set_flow_data_act2` (line 14).

b) Computing a bin offset within a flow marker: To index a bin within a flow marker two values must be added: the flow offset, and the bin offset. While the former can be computed as described in the previous paragraph, the latter is computed by the truncation operator. The quantized packet length passed as an argument to the action responsible for performing truncation (`truncation_act`, line 8) is computed by action `quantization_act` (line 3) using a simple bit shift. Then, the translation between a quantized packet length and the corresponding bin offset can also be computed offline once a specific FMA parameterization is known, and later loaded by the control plane into the truncation table (refer to Section IV-A). Pre-computing these values in the control plane saves stateful memory and pipeline's stages for either monitoring more flows or executing other forwarding behaviors.

```
1  // triggered by the quantization table
2  // bin_width_shift depends on the Quantization Level (QL)
3  action quantization_act(bit<32> bin_width_shift){
4         md.binIndex_quant =
5         (bit<32>) (md.pkt_length >> bin_width_shift);
6  }
7  // triggered by the truncation table
8  action truncation_act(bit<32> new_index, bool flag) {
9         md.bin_offset = new_index;
10        md.trunc_flag = flag;
11 }
12 //triggered by the flow table (FT_2)
13 // compute the offset of the bin in the RG partition
14 action set_flow_data_act2(bit<32> flow_offset) {
15        md.rg_cell_offset = flow_offset + md.bin_offset;
16
17 //triggered after matching the flow table (FT_2)
18 action reg_grid_act2() {
19        bit<16> value;
20        reg_grid2.read(value, md.rg_cell_offset);
21        reg_grid2.write(md.rg_cell_offset, value+1);
22 }
```

Listing 1: P4 code fragment that implements the actions performed per-packet by the FMA. The complexity of the truncation operator and of the computation of flow marker offsets is offloaded to the control plane and the resulting values are loaded through MAUs.

VII. Evaluation

Here, we present our experimental evaluation of FlowLens aimed at analyzing the accuracy of ML-based flow classification tasks and the efficiency of the switch resources usage.

A. Metrics and Methodology

Our experiments aim at identifying a particular class of flows denoted as the *target* class. For instance, when using FlowLens for covert channel detection, the target class can be covert traffic. We assess the quality of FlowLens using the following set of metrics: *accuracy*, i.e., the percentage of flows that were correctly classified in their class, *false positive rate* (FPR), i.e., flows that do not belong to the target class but were erroneously classified as part of the target, and *false negative rate* (FNR), i.e., flows of the target class that were flagged as not belonging to the class. We also resort to related metrics such as *precision* – ratio of the number of relevant flows retrieved to the total number of relevant and irrelevant flows retrieved – and *recall* – ratio of the number of relevant flows retrieved to the total number of relevant flows.

We train our system to be able to identify specific target class flows within the context of three usage scenarios:

Covert channel detection: We train our system to identify Skype flows carrying covert channels encoded by two censorship resistance tools: Facet [38] and DeltaShaper [6]. We train two independent FlowLens applications, for Facet and for DeltaShaper traffic, using a balanced dataset including covert / legitimate samples of recorded flows. The traffic is classified using the XGBoost [7] classifier, based on the packet length distribution of the sampled flows.

Website fingerprinting: We train a second FlowLens application to identify webpages browsed through encrypted tunnels.

Table I. SCALABILITY OF FLOWLENS.

Use Case	FMA Configuration	Marker	Raw Dist.	Scaling
Covert Channels	$\langle QL_{PL}$=4, Top-N=10$\rangle$	20B	3000B	150×
Website Fgpt.	$\langle QL_{PL}$=5$\rangle$	94B	3000B	32×
Botnet Detection	$\langle QL_{PL}$=4, QL_{IPT}=6$\rangle$	302B	10200B	34×

Table II. HARDWARE RESOURCE CONSUMPTION.

Resources	Computational			Memory	
	eMatch xBar	Gateway	VLIW	TCAM	SRAM
Usage	8.46%	5.21%	3.39%	0.00%	38.54%

We leverage the dataset made available by Herrman et al. [29]. This dataset has been widely used for the evaluation of novel website fingerprinting techniques [93, 60], and it contains traces of webpage accesses over OpenSSH. Websites are fingerprinted resorting to the Multinomial Naïve-Bayes classifier [29], which leverages the packet length distribution of the incoming and outgoing data in a connection as features. This classifier also allows us to illustrate how FlowLens can accommodate alternative truncation schemes whenever a given classifier does not return a ranking of feature importance (Section VII-E).

Botnet detection: Our last FlowLens application aims at detecting the presence of botnet chatter among legitimate P2P traffic. We use the dataset produced by Rahbarinia et al. [64], which comprises traffic flows produced by four benign P2P applications (uTorrent, Vuze, Frostwire, and eMule), and two P2P botnets (Waledac and Storm). Malicious flows can be identified by analyzing packet length and inter-packet timing distributions resorting to a Random Forest classifier [64].

We simulate the classification of flows of a given target class in software based on a set of application-specific flow samples. We also configured all the classifiers (Multinomial Naïve-Bayes, XGBoost, and Random Forest) to use the same hyperparameters suggested by the papers we drew our use-cases from. Throughout the evaluation, we assess the performance of different FlowLens configurations while exposing the system to a workload that, to the best of our abilities, mimics those described in the literature. However, we highlight the adoption of a single holdout test instead of the cross-validation approach employed in other representative works [7, 53]. The reason is that, when applying truncation (Section-IV), FlowLens employs a pre-training step to obtain a feature ranking from the classifier. Then, it uses the top-N most important ones to fill the Truncation Table (Figure-4). Since cross-validation returns an average of the results obtained by multiple holdout models trained with different splits of the dataset, the resulting top-N features would not directly translate to be the top-N ones found in a particular model instance, namely in the model to be deployed on the switch for classification. Further, we chose a 50/50 holdout to increase the amount of unseen (test) data and better assess the generalization ability of the classifier.

B. Overall Performance

To give a general insight into the performance of FlowLens, Table I presents the scalability gains of our system when it is used to classify flows for covert channel detection, website fingerprinting, and botnet traffic detection while displaying an accuracy loss of at most 3% when compared with the use of complete packet frequency distributions. For these experiments, we generated the possible combinations of flow markers for the three considered use case scenarios, and assessed whether they allow for accurate flow classification despite their compact size. Packet lengths (PL) vary from 1 to 1500 bytes (MTU), and each cell of a flow marker has a size of 2 bytes.

These results show that, when the quantization and truncation parameters are properly fine-tuned (i.e., QL and truncation table), FlowLens can monitor at least 32 times more flows when compared to the baseline setup without compression, i.e., QL=0 and truncation disabled. Our system can also reach a 150 fold increase in its monitoring capacity when detecting covert channels. This is achieved for QL=4 and by selecting the top-10 most relevant bins for truncation. In this case, with a flow marker as small as 20 bytes, FlowLens manages to achieve a classification accuracy of 93%, only 3% shorter than the result obtained using raw packet length distributions. For website fingerprinting, the flow marker is larger (94 bytes) because we face a multi-class classification problem – different websites are better classified resorting to different bins. Thus, the truncation table is configured to map all quantized packet lengths. Lastly, for botnet chatter detection, we combine the quantization of packet inter-arrival time distribution (IPT) with the PL distribution. In this case, we achieve a marker size of 302 bytes which enables the bookkeeping of $>30\times$ flows.

In general, the absolute number of flows that FlowLens can handle ultimately depends on the switches' available SRAM. The NDA we have signed with Tofino prevents us from disclosing the amount of switch memory but other sources [51] reveal that current switches feature hundreds of MBs of SRAM.

C. Hardware Resource Efficiency

To evaluate the efficiency of FlowLens's hardware resource usage on the switch, we focus independently on the data plane and on the control plane. As for the data plane, Table II shows the average hardware resource consumption of FlowLens across all stages of the switch. The table shows that besides the SRAM required for the tables and register, the consumption of other resources is negligible. Since our flow matching logic entirely relies on exact matching, the FMA's flow table does not consume any of the TCAM resources on the switch. In tandem with the deployment of flow tables in SRAM, FlowLens leaves over 60% of SRAM available. Overall, these results suggest that FlowLens makes enough room for the concurrent execution of many other common forwarding behaviors, like access control, rate limiting or encapsulation, that do not necessarily require an extensive use of the stateful memory in the switch pipeline.

On the control plane, the switch has sufficient resources to fit all models used by FlowLens and to readily classify flows. In particular, the botnet chatter detection is our largest model, occupying only 140MB of memory, and 5.6MB of storage when compressed. In contrast, the model for covert channel detection uses only 64KB of memory and 24KB of storage. All these models comfortably fit within the control plane hardware resources, which has 32GB available RAM. Additionally, the flow classification step is very fast in all cases. For covert channel detection, once the flow markers have been collected from the data plane, the median of the time it takes for the classifier to output a label for a sample flow ranges approximately from 100 to 200 microseconds on the switch's

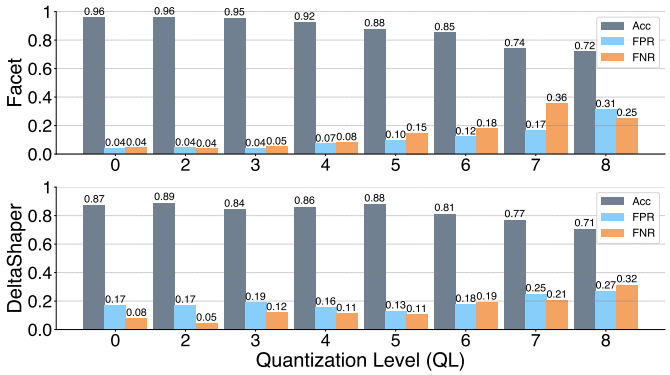


Figure 6. Accuracy, FPR, and FNR for multimedia protocol tunneling detection when using quantized packet length distributions.

Table III. FLOW MARKER SIZE FOR DIFFERENT QUANTIZATION LEVELS.

Bins and Memory	**Quantization Level (QL)**							
`len(1 bin) = 2 Bytes`	0	2	3	4	5	6	7	8
Number of Bins	1500	375	188	94	47	24	12	6
Memory per Flow(B)	3000	750	376	188	94	48	24	12

Intel Broadwell 8-core general-purpose CPU operating at 2 GHz. These results indicate that flow classification can be efficiently conducted on the switch control plane.

Next, we present a set of micro-benchmarks which allow us to assess the benefits of our flow marker generation scheme. We will see that flow marker size (hence memory efficiency) tends to be more sensitive than classification accuracy to small variations in the quantization. The trend is the inverse for truncation, where accuracy is more sensitive to small variations than flow marker size. Our optimizer helps to find sweet-spot setups on the Pareto curve (as explained in Section VII-H).

D. Effects of Quantization

To study the effects of FlowLens's compression schemes, we first focus on the generation of flow markers for packet length distributions and start by analyzing the trade-offs of quantization. We present our main findings:

1. Multimedia covert channels can be detected with up to 92% accuracy using 188-byte flow markers. We leverage XGBoost to classify covert channels [7]. Figure 6 shows how the absolute values obtained for the accuracy, FPR, and FNR of the classifier vary when identifying Facet and DeltaShaper covert channels for different quantization levels (QL). For instance, for quantization level QL=4 FlowLens can correctly identify Facet and DeltaShaper flows with less than 5% and 1% decrease in accuracy, respectively. Table III shows that, for QL=4, a flow marker can be represented in 94 bins instead of a full distribution composed of 1500 bins, amounting to an order of magnitude memory savings. While a single flow marker is then represented using 188B instead of 3000B, DeltaShaper classification scores are maintained with respect to those obtained when using full information (see Figure 6).

2. Accuracy of website fingerprinting is maintained when compressing flow markers by two orders of magnitude. For assessing the quality of FlowLens on website fingerprinting, we use the Multinomial Naïve-Bayes classifier [29]. We reproduced the multiclass closed-world website fingerprinting task for

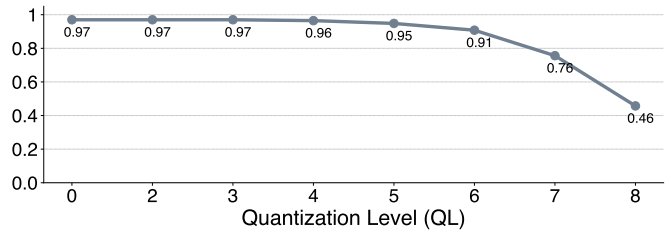


Figure 7. Accuracy results for website fingerprinting when using quantized packet length distributions.

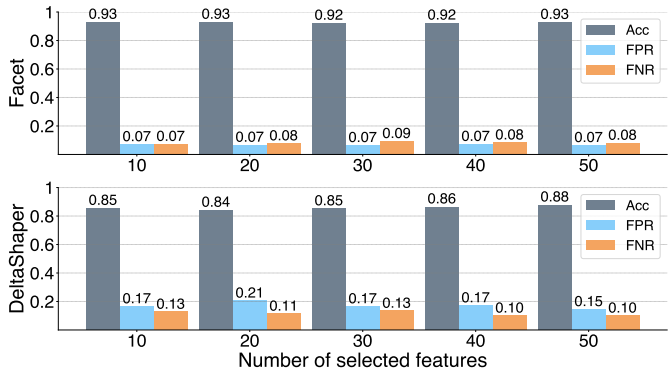


Figure 8. Accuracy, FPR, and FNR for covert channel detection with an increasing number of features for quantization level QL=4.

different quantization levels. Figure 7 shows that FlowLens is able to maintain the same classification accuracy up to a quantization level QL=3. Providing that classification accuracy can be relaxed in favor of memory savings, quantization can be further increased to QL=6, while still achieving over 90% accuracy and reducing a flow marker's memory footprint by two orders of magnitude.

3. Very coarse-grained flow markers are unsuitable for performing traffic differentiation. Figure 7 shows that flow markers can only be compressed to a given factor before causing a steep decrease in the quality of the models' predictions. For instance, in Figure 6, it is possible to observe that for QL=7 the accuracy of the classifier is already over 20% and 10% away from the result obtained with full information for Facet and DeltaShaper, respectively. Thus, it is imperative to find the correct balance between memory savings and accuracy. FlowLens balances this trade-off, for different use cases, through a parameterization during the profiling phase.

E. Effects of Truncation

The second mechanism to generate compact flow markers is that of truncating the flow marker to a subset of bins which make up for the most relevant features leveraged by the classifier. This is illustrated next, as we highlight our main findings after applying tailored truncation in different use cases.

1. Accurate detection of covert channels can be achieved using a flow marker of just 20 bytes. We elaborated a tailored truncation approach based on the importance of features computed by XGBoost. Figure 8 depicts the results obtained when performing quantization with QL=4 and truncating to the top-N most important features. The accuracy, FPR, and FNR rate of the classifier are practically identical when using the

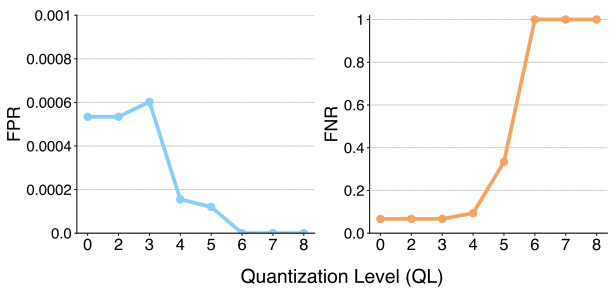


Figure 9. FPR and FNR for `www.amazon.com` at different quantization levels (QL). FNR shows the probability of identifying `www.amazon.com` as some other website. FPR shows the probability of some other website being mistakenly classified as `www.amazon.com`. Truncation is applied at each QL.

Table IV. NUMBER OF BINS USED IN `www.amazon.com` TRUNCATION.

Bins and Memory	**Quantization Level (QL)**							
`len(1 bin) = 2 Bytes`	0	2	3	4	5	6	7	8
Total Number of Bins	1500	375	188	94	47	24	12	6
Bins After Truncation	159	159	156	87	46	23	12	6

top-10 and top-50 features to classify flows (e.g., a difference of only 1% in FNR for Facet flows), and very similar to the results obtained when using full information (refer to QL=4 in Figure 6). Thus, truncation can not only maintain high accuracy, but further reduce the flow marker footprint from 188B (QL=4) to just 20B (QL=4, top-N=10).

2. 20-byte flow markers enable tracking 150× more flows. Covert flow markers can be reduced to just 20B using truncation. This corresponds to a 150× space-saving when representing a flow (from 3000B to 20B). The space freed by compressing a single flow represents an increase in FlowLens's measurement capacity by two orders of magnitude.

3. Fingerprinting accesses to a website yields good results even when feature ranking is unavailable. The truncation method employed for covert channel detection is only applicable when considering classifiers able to output feature importance. To overcome the fact that Herrmann et al.'s [29] classifier is unable to output a rank of feature importance, we perform manual bin selection aimed at identifying a single website, e.g., `www.amazon.com`. Essentially, we first take a collection of access traces performed over a period of time to that particular website. Then, we simply discard the bins that correspond to packet lengths which have had zero counts of the sampled flows. Based on this selection, we then train our classifier accordingly. We can see in Figure 9 that the results obtained using this approach remain competitive. For instance, with quantization level QL=4, flows can be correctly identified with a 0.016% FPR and 9.333% FNR. As shown in Table IV, this flow marker footprint is not as small as with covert channel detection. Yet, it is practical to fingerprint website accesses with QL=4, yielding flow markers with a compression ratio of 1500:87, i.e., 17.2×.

F. Measuring Inter-Packet Timing

In this section, we concentrate on the ability of FlowLens to perform tasks that require both the inspection of packets' inter-arrival (IPT) and length (PL) distributions. To this end, we evaluate FlowLens in detecting P2P botnet chatter. Since the network traffic produced by bots tends to be stealthy and spread

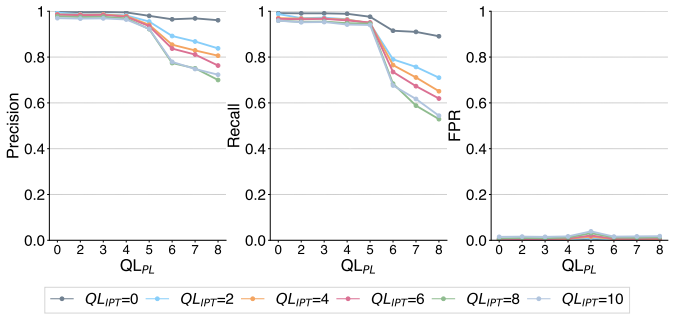


Figure 10. Precision, recall, and FPR for malicious P2P traffic.

across time, packets sent in bot conversations are expected to have a higher IPT than those of legitimate P2P conversations. A conversation consists of the set of flows between any two hosts within a given time window, called *flowgap*. We resort to the Random Forest classifier originally employed by Narang et al. in PeerShark [53], and follow their recommendation to set flowgap to 3600s. Since the largest flowgap is set to 3600s, we vary the quantization of inter-arrival time down to a minimum of 4 bins ($QL_{IPT} = 10$).

Figure 10 depicts the precision, recall, and FPR obtained by the classifier when identifying botnet chatter for different QL applied to both PL and IPT distributions. While $QL_{PL} \leq 4$ slightly degrades precision and recall, we observe a sharp drop in both metrics when $QL_{PL} > 4$. The FPR, however, is not significantly affected by increasing quantization levels. Our experiments also reveal that precision and recall in the identification of legitimate P2P traffic are largely unaffected by the effect of quantization, whereas FPR takes a sharp increase for $QL_{PL} \geq 6$ (20% at $QL_{PL} = 6$ up to 50% at $QL_{PL} = 8$).

This figure also shows that it is possible to accurately identify botnet traffic with compact flow markers. For instance, $\langle QL_{PL} = 4, QL_{IPT} = 6 \rangle$ achieves a recall of 0.96, only 3% worse when compared to the result obtained when using full information (0.99). This accounts for a memory saving of 16× when storing a flow's packet length distribution, as well as occupying just 57 buckets×2B = 114B to keep an inter-packet timing distribution. These results suggest that FlowLens can offer different space-saving/accuracy trade-offs.

G. Performance of Automatic Profiling

To evaluate FlowLens's automatic profiling mechanism, we explore the parameter search space for each use case. For Facet and DeltaShaper, the search space includes the quantization and truncation parameters studied above (48 configurations). For website fingerprinting, the search space corresponds to 8 quantization configurations. For botnet detection, we consider 40 possible configurations based on packet length and IPT quantization, as we refrain from considering those whose $QL_{IPT} = 0$. We configure the optimizer to explore i=10 configurations for covert channel and botnet detection, and i=4 for website fingerprinting. For simplicity, we use no initial sampling for bootstrapping the optimizer, but techniques like Latin Hypercube sampling [75] may be also plugged in.

Fully automatic mode: Table V depicts the results obtained by our automatic profiler to choose an FMA configuration. In all cases, the profiler chooses a configuration that, albeit not the best accuracy wise, still provides a competitive accuracy

Table V. RESULTS OF THE PROFILING PROCEDURE, INCLUDING THE CONFIGURATION OUTPUT BY THE OPTIMIZER AND THE TOP-3 EXPLORED CONFIGURATIONS (LISTED BY DECREASING ACCURACY, EXCEPT FOR THE CASE OF BOTNETS WHICH CORRESPONDS TO MALICIOUS TRAFFIC RECALL).

Config. Rank	Facet (i=10)	DeltaShaper (i=10)	Website Fingerprinting (i=4)	Botnets (i=10)
#1	⟨QL=2, Top-N=all⟩ = 0.960	⟨QL=5, Top-N=all⟩ = 0.880	⟨QL=0⟩ = 0.970	⟨QL_{PL}=2, QL_{IPT}=2⟩ = 0.970
#2	⟨QL=3, Top-N=50⟩ = 0.951	⟨QL=0, Top-N=all⟩ = 0.873	⟨QL=4⟩ = 0.965	⟨QL_{PL}=0, QL_{IPT}=6⟩ = 0.969
#3	⟨QL=0, Top-N=30⟩ = 0.947	⟨QL=0, Top-N=20⟩ = 0.870	⟨QL=5⟩ = 0.948	⟨QL_{PL}=4, QL_{IPT}=6⟩ = 0.960
Output	⟨QL=3, Top-N=10⟩ = 0.944	⟨QL=5, Top-N=10⟩ = 0.840	⟨QL=4⟩ = 0.965	⟨QL_{PL}=3, QL_{IPT}=1024⟩ = 0.953

while generating compact flow markers. For instance, for Facet, the top-3 configurations exhibit a marker size of 375, 50, and 30 bins, respectively. Our profiler chooses a configuration that provides a marker size of 10 bins while achieving an accuracy only 1.6% worse than the configuration with the best-found accuracy (and with a 37× smaller marker). Our reward policy leads the optimizer to perform good decisions over the explored configurations. In website fingerprinting, the profiler outputs the top-2 configuration rather than top-1 since the latter's marker size is too big in comparison (1500 vs 94 bins). The profiler also refrains from choosing top-3, a configuration whose marker is 2× smaller but less accurate. This trend can be observed for the remaining use cases.

Smaller marker for target accuracy: FlowLens can find a configuration that exceeds a minimum accuracy threshold, and that provides the smallest marker. For instance, we set a target accuracy of 0.85 for a DeltaShaper configuration. Among the 10 experimented configurations, the optimizer has found 3 candidate configurations with an accuracy larger than the set threshold. The system output ⟨QL=4, Top-N=30⟩ = 0.850, albeit finding ⟨QL=5, Top-N=40⟩ = 0.876 or ⟨QL=0, Top-N=40⟩ = 0.880, two other configurations which produced larger accuracy at the expense of a larger marker.

Best accuracy given a size constraint: The system is also able to find configurations with a larger accuracy value, given a maximum marker size. Additionally, and since the size of a marker can be computed offline without first trying a configuration, we achieve a reduction in the search space. In the case of DeltaShaper, setting a maximum marker size equal to 30 enables the reduction of the search space from 48 to 21 possible configurations. In this case, the optimizer outputs ⟨QL=2, Top-N=30⟩ = 0.890, albeit finding other smaller but less accurate alternatives such as ⟨QL=3, Top-N=20⟩ = 0.850.

H. Comparison with Related Approaches

In this section, we compare FlowLens against two related approaches: i) techniques which are able to produce compressed representations of packet distributions, and ii) techniques for collection of traffic features resorting to programmable switches.

Alternative feature compression approaches: Online Sketching (OSK) [22] and Compressive Traffic Analysis (CTA) [55] generate compressed packet length/inter-packet timing distributions using linear transformations. However, both approaches depend on matrix multiplications and/or floating-point operations unsupported by current switching hardware. Yet, we compare the classification accuracy of FlowLens against the accuracy obtained by OSK and CTA when using each technique to compress flow representations.

For evaluating the quality of the solutions yielded by the different compression techniques, we leverage the concept of

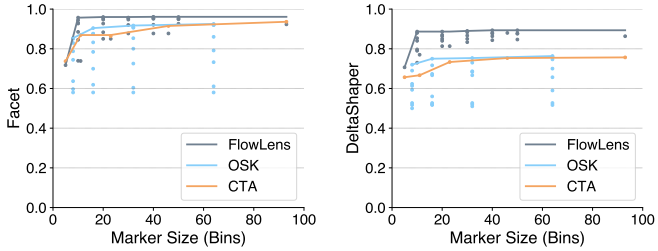


Figure 11. Pareto frontier for covert channel detection when using FlowLens, OSK, and CTA. Dots show individual configurations.

Pareto optimality [57] which allows us to compare possible solutions to multi-criteria optimization problems (flow marker size vs. accuracy, in our case). A solution is said to be Pareto optimal if it cannot be improved in one of the objectives without adversely affecting the other. By generating the set of all of the potentially optimal solutions (Pareto frontier) for each approach, we can observe which approach delivers the best trade-offs between classification accuracy and marker size.

Figure 11 depicts the accuracy obtained in the classification of covert channels when using flow markers (with size up to 100 bins) generated by FlowLens, OSK, and CTA, while using different compression ratios. FlowLens configurations are achieved by combining the different quantization and truncation parameters. Solid lines represent the Pareto frontiers [44] that capture the best configurations for the three approaches. Overall, FlowLens produces flow markers that exhibit a better accuracy/memory trade-off and obtain the most accurate compressed representations of flows. For instance, in DeltaShaper, most FlowLens configurations achieve over 0.80 accuracy (and a maximum of 0.89 using a flow marker with a size of only 10 bins). In comparison, the most accurate OSK marker takes 16 bins and achieves an accuracy of only 0.76. A similar trend occurs in the case of Facet detection.

Alternative feature collection approaches: Systems such as *Flow [74] are able to collect fine-grained packet features at line rate from the switch and offload them to dedicated servers, where the packet distributions can be computed and analyzed by other dedicated systems for specific applications. FlowLens provides a complementary decentralized design where both the collection of packet distribution features and the application-specific analysis (i.e., flow classification) take place on the switches, thus achieving considerable savings in communication, compute, and storage hardware resources.

To estimate the potential gains of our design, we analyze the communication costs of both *Flow and FlowLens. Assuming the existence of 250k concurrent flows where each flow sends 15k packets during a collection window of 30 seconds, *Flow offloads data structures named *grouped packet vectors* (GPVs), each containing a flow key and a list of packet lengths, from a sequence of packets in a flow, on an average of 640ms [74]

which totals 47 evictions. Since each GPV has a fixed header of 24 bytes, assuming 2 bytes to encode a packet length, *Flow must transfer (24B×47 + 2B×15k)×250k = 7.78GB per collection window. This data would then need to be processed on a dedicated server. In contrast, FlowLens only transfers the classification score of each flow at the end of the collection window which involves sending a fixed-size header per flow (13B for flow ID plus a 4B score value) times 250k flows, i.e., ≈4.25MB. Thus, FlowLens exhibits a communication footprint three orders of magnitude smaller than *Flow.

VIII. Security Analysis

We now analyze the security properties of FlowLens when functioning under an adversarial model. The overarching goal of the adversary is to be able to generate flows of a target application class without being detected by FlowLens. We consider three categories of increasingly sophisticated adversaries considering their knowledge about FlowLens and the models employed in ML-based security applications.

1. No knowledge about FlowLens nor the ML model: In the weakest threat model, the attacker knows nothing about the presence of FlowLens in the network infrastructure, nor the details of the models being used by the ML-based security applications leveraging the capabilities of our system. In such a case, as shown in the sections above, ML-based security applications making use of the vanilla FlowLens setup can identify different target classes of traffic with high accuracy.

2. FlowLens-aware adversary: In the second case, we consider an adversary that is aware of the deployment of FlowLens in the network infrastructure, but who is unaware of the particular machine learning models being used to filter the network for particular classes of traffic. In this case, the adversary may attempt to launch two particular types of attacks:

Flow aggregation attacks: An adversary may attempt to evade FlowLens's classifier by misusing the truncation and quantization steps to make the aggregation of flows of a given class of traffic indistinguishable from another class. In this sense, this type of attack is similar to our covert channel scenario (Section VII-D) where the adversary's goal is to mimic the distribution of legitimate traffic and evade a classifier. Figure 6 and Figure 8 show that a finer-grained aggregation of packet distributions does make it harder to evade the classifier. This suggests that increasing flow marker granularity makes FlowLens more robust against flow aggregation attacks.

Evading collection windows: When analyzing long-lived network flows, FlowLens collects flow markers during a maximum pre-defined collection window. Once this window elapses, the FMA located on a given switch stops monitoring flows while the flow markers are read and FMA data structures are reset. An adversary may attempt to exploit this window of opportunity to transmit a class of traffic targeted by FlowLens during this period. However, as mentioned in Section IV-D, FlowLens can tolerate such attacks provided that multiple switches are used in an interleaved fashion to ensure that at least one switch can collect traffic pertaining to flows traversing the network.

DoS attacks: A FlowLens-aware adversary may also attempt to compromise the availability of our system. For instance, it may try to mount a DoS attack based on the transmission of packets with random IP addresses, forcing FlowLens to keep track of multiple dummy flows and waste the switch memory. To mitigate such a threat, FlowLens can temporarily prevent the installation of new rules in the FMA flow table when it detects unusual bursts of traffic, or reconfigure FMA parameters on-the-fly to store smaller (yet less accurate) flow signatures so as to increase the number of measured flows.

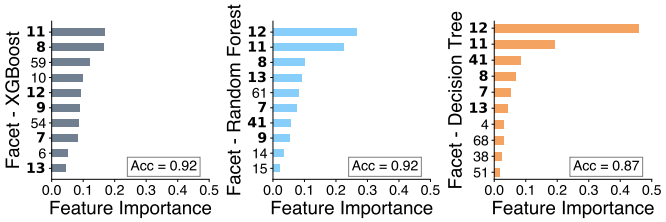


Figure 12. Features (PL bins) collected by the FMA for the ⟨QL=4, top-N=10⟩ configuration, when considering different classifiers to identify Facet covert channels. Features in bold are shared among at least two classifiers.

3. FlowLens and ML model-aware adversary: The third adversary we consider is cognizant of the operation of FlowLens and knowledgeable about the ML model used by a given ML-based application. Apart from an adversary's attempts to evade or compromise the availability of our system, such an adversary aims to leverage adversarial ML techniques [3] to subvert the correct behavior of the model. These attacks can be grouped in two main categories [3]: i) *training-time*, where an attacker aims at manipulating the training set used by the ML model through the insertion of specific samples that alter the decision boundaries of the classifier; ii) *test-time*, where an attacker aims to evade classifiers by crafting traffic samples in such a way that these fool the classifier during its operational phase.

In general, providing defenses to such attacks is orthogonal to FlowLens's ability to collect flow markers and it concerns the particular models used by the different ML-based security applications. Nevertheless, FlowLens is compatible with various techniques aimed at increasing the robustness of the models used for traffic analysis. For mitigating *training-time* attacks, FlowLens's profiling phase can incorporate mechanisms aimed at filtering out contaminated instances upon training [45, 83, 25, 67]. Alternatively, FlowLens operators can leverage recent models whose training is explicitly hardened against the introduction of adversarial samples [16, 15, 78, 34]. For tackling *execution-time* attacks, FlowLens is compatible with the use of several techniques that increase the difficulty of an adversary to successfully evade network traffic classifiers. For instance, FlowLens can leverage classifier ensembles [1, 11, 42] or randomize the classifiers deployed at test-time [50].

Hardening FlowLens against adversarial ML attacks: We performed a simple experiment to understand whether FlowLens can leverage the above techniques to improve its robustness to adversarial attacks, while still collecting flow markers of small size. To this end, we profiled three different classifiers – XGBoost, Random Forest, and Decision Tree [7] – to identify Facet traffic in a ⟨QL=4, top-N=10⟩ FMA configuration.

Figure 12 depicts the importance of the top 10 features selected by the different classifiers after FlowLens's profiling step. Recall that, for a QL=4, there is a total of 94 features (bins), from which only the top-10 is considered. We draw two main observations from this figure. First, since all three classifiers share several features (marked in bold), crafting the

traffic to subvert a given feature (e.g., feature 12), requires extra effort to collectively assess how it affects the classification accuracy not just of a single, but of all three classifiers. Second, each classifier selects a subset of features that are exclusive to it. Thus, while an adversary may shape a given flow to respect the features analyzed by a particular classifier, there may be another classifier that considers a different set of features. For instance, XGBoost leverages bins 59, 10, 54, and 6 to better inform a prediction, while the Random Forest classifier ignores these features and includes 61, 14, 15 in its top-10 instead.

To run multiple classifiers in execution-time, FlowLens must collect a superset of all meaningful features required by each model. Thus, it is expected that flow markers will increase their size. Figure 12 suggests that randomization, i.e., the random selection of one possible classifier, can provide a good compromise between robustness to adversarial ML and flow marker size. While a flow marker consists of 10 features for a given classifier (amounting to 20B), a flow marker that enables FlowLens to choose from three different models to classify flows uses a total of 18 distinct features, producing a flow marker amounting to just 36B. In a similar fashion, FlowLens could use all three classifiers to produce an ensemble which will ultimately classify a flow by majority voting [11].

Performance impact of the defense mechanisms: Although we have not empirically assessed the performance overheads caused by the proposed defense mechanisms, we argue that these mechanisms should not significantly impair the performance of FlowLens. Nevertheless, we reckon that they may require additional resources. The impact on resource allocation could be estimated, e.g., by measuring the memory consumed using ensembles, or by studying how many switches would suffice to plummet the risks of window evasion attacks.

IX. Related Work

There is a considerable body of work proposing approaches for building efficient network telemetry systems for large scale networks [87]. Programmable switches can leverage TCAM-based flow tables for keeping flow data [80, 76, 52] and wildcard rules [13] to record a few statistics about a given flow [46]. The major drawback of this technique is tied to the limited size of TCAM which prevents the bookkeeping of more than a few thousand flows [88]. While multiple flows can be combined in the same table entry [91, 52], this aggregation jeopardizes the accurate representation of a large number of flows [88].

Traffic sampling techniques enable the collection of statistics for a large set of flows by recording a small number of packets of each flow [17, 56, 23]. Examples of general monitoring systems implementing sampling are OpenSample [77] or Planck [65]. Canini et al. [17] introduced a per-flow measurement technique that holds on the partial sampling of flows. However, the accuracy of sampling techniques is usually reduced when one aims at obtaining a faithful representation of a flow's distribution [39]. Moreover, increasing the sample rate is at odds with a larger memory footprint which can impact the overall performance of the network [30].

Probabilistic data structures known as sketches enable the error-bounded representation of flows' statistics within restrictive memory limits [88]. While multiple sketches allow for the extraction of flow's coarse-grained features [21, 86, 39, 43, 31, 85, 32], their strive for generality prevents recording fine-grained information such as approximations of flows' packet lengths and timing distributions. NetWarden [82] uses sketches to record approximate distributions of inter-packet timing distributions for a specific security task. In contrast, FlowLens can be broadly applicable to a range of ML-based applications. Coskun et al. [22] and Nasr et al. [55] explore additional ways to compress packet distributions based on the use of linear projections. Unfortunately, such techniques cannot be implemented efficiently in current switching hardware.

Recent systems relying on network query refinement [54], such as Turboflow [73] and *Flow [74], allow the data plane to offload simple packet features to servers for aggregation and processing. However, differently from FlowLens, such a strategy may increase the risks of network congestion and introduce scalability bottlenecks in large networks [41, 87].

X. Conclusions

This work proposed FlowLens, the first traffic analysis system for ML-based security applications that collects and analyses compact representations of flows' packet distributions – flow markers – within programmable switches. We evaluated our system for three use cases comprising the detection of network covert channels, website fingerprinting, and botnet chatter detection. FlowLens can accurately predict these classes of traffic flows with the help of compact flow markers, allowing for a reduction between one to two orders of magnitude of the memory footprint to represent packet distributions.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the SFRH/BD/136967/2018 grant, and the PTDC/EEI-COM/29271/2017, UIDB/50021/2020, and PTDC/CCI-INF/30340/2017 (uPVN) projects.

References

[1] J. Aiken and S. Scott-Hayward, "Investigating adversarial attacks against network intrusion detection systems in sdns," in *Proceedings of the Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Dallas, TX, USA, November 2019, pp. 1–7.

[2] B. Anderson and D. McGrew, "Identifying encrypted malware traffic with contextual flow data," in *Proceedings of the 2016 ACM workshop on artificial intelligence and security*, Vienna, Austria, October 2016, pp. 35–46.

[3] G. Apruzzese, M. Colajanni, L. Ferretti, and M. Marchetti, "Addressing adversarial attacks against security systems based on machine learning," in *2019 11th International Conference on Cyber Conflict*, vol. 900, Tallinn, Estonia, May 2019, pp. 1–18.

[4] Barefoot Networks, Tofino Switch https://www.barefootnetworks.com/products/brief-tofino/, accessed: 2021-01-05.

[5] ——, P4 Studio, https://www.barefootnetworks.com/products/brief-p4-studio/, accessed: 2021-01-05.

[6] D. Barradas, N. Santos, and L. Rodrigues, "Deltashaper: Enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams," in *Proceedings on Privacy Enhancing Technologies*, Minneapolis, MN, USA, July 2017, pp. 5–22.

[7] ——, "Effective detection of multimedia protocol tunneling using machine learning," in *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA, August 2018, pp. 169–185.

[8] D. Barradas and S. Signorello, "FlowLens code repository," https://github.com/dmbb/FlowLens, 2020, accessed: 2021-01-05.

[9] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

[10] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on Machine Learning*, Atlanta, GA, USA, June 2013, pp. 115–123.

[11] B. Biggio, G. Fumera, and F. Roli, "Multiple classifier systems for robust classifier design in adversarial environments," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 27–41, 2010.

[12] 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, July 2014.

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[14] Broadcom, Tomahawk II 6.4Tbps Ethernet Switch, https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch, accessed: 2021-01-05.

[15] S. Calzavara, C. Lucchese, and G. Tolomei, "Adversarial training of gradient-boosted decision trees," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, Beijing, China, November 2019, pp. 2429–2432.

[16] S. Calzavara, C. Lucchese, G. Tolomei, S. Abebe, and S. Orlando, "Treant: training evasion-aware decision trees," *Data Mining and Knowledge Discovery*, pp. 1–31, 2020.

[17] M. Canini, D. Fay, D. Miller, A. Moore, and R. Bolla, "Per flow packet sampling for high-speed network monitoring," in *Proceedings of the First IEEE International Communication Systems and Networks and Workshops*, Chennai, India, December 2009, pp. 1–10.

[18] C. Cascaval and D. Daly, P4 Architectures, https://p4.org/assets/p4-ws-2017-p4-architectures.pdf, accessed: 2021-01-05.

[19] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Los Angeles, CA, USA, August 2017, pp. 1–14.

[20] Cisco, Cisco Encrypted Traffic Analytics Whitepaper, https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf, accessed: 2021-01-05.

[21] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219–1232, 2005.

[22] B. Coskun and N. Memon, "Online sketching of network flows for real-time stepping-stone detection," in *Proceedings of the IEEE Annual Computer Security Applications Conference*, Honolulu, HI, USA, December 2009, pp. 473–483.

[23] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003, pp. 325–336.

[24] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[25] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, "On the (statistical) detection of adversarial examples," *arXiv preprint arXiv:1702.06280*, 2017.

[26] J. Gu, J. Wang, Z. Yu, and K. Shen, "Walls have ears: Traffic-based side-channel attack in video streaming," in *Proceedings of the IEEE Conference on Computer Communications*, Honolulu, HI, USA, April 2018, pp. 1538–1546.

[27] R. Habeeb, F. Nasaruddin, A. Gani, I. Hashem, E. Ahmed, and M. Imran, "Real-time big data processing for anomaly detection: A survey," *International Journal of Information Management*, vol. 45, pp. 289–307, 2019.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[29] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier," in *Proceedings of the ACM Workshop on Cloud Computing Security*, Chicago, IL, USA, November 2009, pp. 31–42.

[30] Q. Huang, X. Jin, P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, August 2017, pp. 113–126.

[31] Q. Huang, P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Budapest, Hungary, August 2018, pp. 576–590.

[32] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in *Proceedings*

of the 15th International Conference on Emerging Networking Experiments And Technologies, Orlando, FL, USA, December 2019, pp. 285–291.

[33] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable in-network security for context-aware BYOD policies," in *Proceedings of the 29th USENIX Security Symposium*, Virtual Event, USA, August 2020, pp. 595–612.

[34] A. Kantchelian, J. Tygar, and A. Joseph, "Evasion and hardening of tree ensemble classifiers," in *Proceedings of the 33rd International Conference on Machine Learning,*, vol. 48, New York, NY, USA, June 2016, pp. 2387–2396.

[35] S. Khattak, N. Ramay, K. Khan, A. Syed, and S. Khayam, "A taxonomy of botnet behavior, detection, and defense," *IEEE communications surveys & tutorials*, vol. 16, no. 2, pp. 898–924, 2013.

[36] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi, "Enabling event-triggered data plane monitoring," in *Proceedings of the Symposium on SDN Research*, San Jose, CA, USA, March 2020, pp. 14—26.

[37] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan, "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering," in *Proceedings of the 27th IEEE international conference on network protocols*, Chicago, IL, USA, October 2019, pp. 1–12.

[38] S. Li, M. Schliep, and N. Hopper, "Facet: Streaming over videoconferencing for censorship circumvention," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, Scottsdale, AZ, USA, November 2014, pp. 163–172.

[39] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, Santa Clara, CA, USA, March 2016, pp. 311–324.

[40] M. Liberatore and B. N. Levine, "Inferring the source of encrypted http connections," in *Proceedings of the 13th ACM conference on Computer and Communications Security*, Alexandria, VA, USA, October 2006, pp. 255–263.

[41] Y. Lin, Y. Zhou, Z. Liu, K. Liu, Y. Wang, M. Xu, J. Bi, Y. Liu, and J. Wu, "Netview: Towards on-demand network-wide telemetry in the data center," *Computer Networks*, vol. 180, 2020.

[42] L. Liu, W. Wei, K.-H. Chow, M. Loper, E. Gursoy, S. Truex, and Y. Wu, "Deep neural network ensembles against deception: Ensemble diversity, accuracy and robustness," in *Proceedings of the 16th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, Monterey, CA, USA, November 2019, pp. 274–282.

[43] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communications Conference*, Florianópolis, Brazil, August 2016, pp. 101–114.

[44] D. Luc, *Pareto Optimality*. Springer New York, 2008, pp. 481–515.

[45] S. Ma and Y. Liu, "Nic: Detecting adversarial samples with neural network invariant checking," in *Proceedings of the 26th Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2019.

[46] M. Malboubi, L. Wang, C.-N. Chuah, and P. Sharma, "Intelligent sdn based traffic (de) aggregation and measurement paradigm (istamp)," in *Proceedings of the IEEE Conference on Computer Communications*, Toronto, Canada, April 2014, pp. 934–942.

[47] R. McPherson, A. Houmansadr, and V. Shmatikov, "CovertCast: Using live streaming to evade internet censorship," *Proceedings on Privacy Enhancing Technologies*, vol. 2016(3), pp. 212–225, 2016.

[48] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici, "N-baiot—network-based detection of iot botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[49] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev, "Nethide: Secure and practical network topology obfuscation," in *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA, August 2018, pp. 693–709.

[50] D. Meng and H. Chen, "Magnet: a two-pronged defense against adversarial examples," in *Proceedings of the ACM SIGSAC conference on computer and communications security*, Dallas, TX, USA, October 2017, pp. 135–147.

[51] 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*, Los Angeles, CA, USA, August 2017, pp. 15–28.

[52] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 419–430, 2015.

[53] P. Narang, S. Ray, C. Hota, and V. Venkatakrishnan, "Peershark: detecting peer-to-peer botnets by tracking conversations," in *Proceedings of the IEEE Security and Privacy Workshops*, San Jose, CA, USA, May 2014, pp. 108–115.

[54] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Los Angeles, CA, USA, August 2017, pp. 85–98.

[55] M. Nasr, A. Houmansadr, and A. Mazumdar, "Compressive traffic analysis: A new paradigm for scalable traffic analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, October 2017, pp. 2053–2069.

[56] Netflow, https://www.ietf.org/rfc/rfc3954.txt, accessed: 2021-01-05.

[57] P. Ngatchou, A. Zarei, and A. El-Sharkawi, "Pareto multi objective optimization," in *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, Arlington, VA, USA, January 2005, pp. 84–91.

[58] T. Nguyen and G. Armitage, "A survey of techniques for

internet traffic classification using machine learning." *IEEE Communications Surveys and Tutorials*, vol. 10, no. 1-4, pp. 56–76, 2008.

[59] T. O'Connor, R. Mohamed, M. Miettinen, W. Enck, B. Reaves, and A.-R. Sadeghi, "Homesnitch: Behavior transparency and control for smart home iot devices," in *Proceedings of the 12th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Miami, FL, USA, May 2019, pp. 128—-138.

[60] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, Chicago, IL, USA, October 2011, pp. 103–114.

[61] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the V out of {NFV}," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, November 2016, pp. 203–216.

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python ," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[63] C. Putman, Abhishta, and L. Nieuwenhuis, "Business model of a botnet," in *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Cambridge, UK, March 2018, pp. 441–445.

[64] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, "Peerrush: Mining for unwanted p2p traffic," *Journal of Information Security and Applications*, vol. 19, no. 3, pp. 194–208, 2014.

[65] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 407–418, 2014.

[66] A. Reed and M. Kranch, "Identifying https-protected netflix videos in real-time," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, Scottsdale, AZ, USA, March 2017, pp. 361–368.

[67] K. Roth, Y. Kilcher, and T. Hofmann, "The odds are odd: A statistical test for detecting adversarial examples," in *Proceedings of the 36th International Conference on Machine Learning*, vol. 97, Long Beach, CA, USA, June 2019.

[68] R. Schuster, V. Shmatikov, and E. Tromer, "Beauty and the burst: Remote identification of encrypted video streams," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, Canada, August 2017, pp. 1357–1374.

[69] N. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, March 2017, pp. 67–82.

[70] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Florianópolis, Brazil, August 2016, pp. 15–28.

[71] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the ACM Symposium on SDN Research*, Santa Clara, CA, USA, April 2017, pp. 164–176.

[72] J. Snoek, H. Larochelle, and R. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, vol. 2, Lake Tahoe, NV, USA, December 2012, pp. 2951–2959.

[73] J. Sonchack, A. Aviv, E. Keller, and J. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, Porto, Portugal, April 2018, pp. 1–16.

[74] J. Sonchack, O. Michel, A. Aviv, E. Keller, and J. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, USA, July 2018, pp. 823–835.

[75] M. Stein, "Large sample properties of simulations using latin hypercube sampling," *Technometrics*, vol. 29, no. 2, pp. 143–151, 1987.

[76] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Flowcover: Low-cost flow monitoring scheme in software defined networks," in *Proceedings of the IEEE Global Communications Conference*, Austin, TX, USA, December 2014, pp. 1956–1961.

[77] J. Suh, T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*, Madrid, Spain, June 2014.

[78] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," in *Proceedings of the Sixth International Conference on Learning Representations*, Vancouver, Canada, April 2018.

[79] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home device events," in *Proceedings of the 27th Network and Distributed Security Symposium*, San Diego, CA, USA, February 2020.

[80] N. van Adrichem, C. Doerr, and F. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *Proceedings of the IEEE Network Operations and Management Symposium*, Krakow, Poland, May 2014.

[81] C. Wright, S. Coull, and F. Monrose, "Traffic morphing: An efficient defense against statistical traffic analysis," in *Proceedings of the 16th Annual Network & Distributed Security Symposium*, San Diego, CA, USA, February 2009.

[82] J. Xing, Q. Kang, and A. Chen, "Netwarden: Mitigating

network covert channels while preserving performance," in *Proceedings of the 29th USENIX Security Symposium*, Virtual Event, USA, August 2020, pp. 2039–2056.

[83] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," in *Proceedings of the 25th Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2018.

[84] J. Yan and J. Kaur, "Feature selection for website fingerprinting," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 200–219, 2018.

[85] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Budapest, Hungary, August 2018, pp. 561–575.

[86] T. Yang, L. Wang, Y. Shen, M. Shahzad, Q. Huang, X. Jiang, K. Tan, and X. Li, "Empowering sketches with machine learning for network measurements," in *Proceedings of the 2018 ACM SIGCOMM Workshop on Network Meets AI & ML*, Budapest, Hungary, August 2018, pp. 15–20.

[87] M. Yu, "Network telemetry: towards a top-down approach," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 1, pp. 11–17, 2019.

[88] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang, "Countmax: A lightweight and cooperative sketch measurement for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2774–2786, December 2018.

[89] F. Zane, G. Narlikar, and A. Basu, "Coolcams: Power-efficient tcams for forwarding engines," in *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, San Francisco, CA, USA, March 2003, pp. 42–52.

[90] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of the 27th Network and Distributed Security Symposium*, San Diego, CA, USA, February 2020.

[91] G. Zhao, H. Xu, S. Chen, L. Huang, and P. Wang, "Deploying default paths by joint optimization of flow table and group table in sdns," in *Proceedings of the 25th IEEE International Conference on Network Protocols*, Toronto, ON, Canada, Ocotober 2017, pp. 1–10.

[92] S. Zhao, M. Chandrashekar, Y. Lee, and D. Medhi, "Real-time network anomaly detection system using machine learning," in *Proceedings of 11th International Conference on the Design of Reliable Communication Networks*, Kansas City, MO, USA, March 2015, pp. 267–270.

[93] Z. Zhuo, Y. Zhang, Z.-l. Zhang, X. Zhang, and J. Zhang, "Website fingerprinting attack on anonymity networks based on profile hidden markov model," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1081–1095, 2017.

pHeavy: Predicting Heavy Flows in the Programmable Data Plane

Xiaoquan Zhang, Lin Cui, *Member, IEEE*, Fung Po Tso, *Senior Member, IEEE*, and Weijia Jia, *Fellow, IEEE*

***Abstract*—Since heavy flows account for a significant fraction of network traffic, being able to predict heavy flows has benefited many network management applications for mitigating link congestion, scheduling of network capacity, exposing network attacks and so on. Existing machine learning based predictors are largely implemented on the control plane of Software Defined Networking (SDN) paradigm. As a result, frequent communication between the control and data planes can cause unnecessary overhead and additional delay in decision making. In this paper, we present *pHeavy*, a machine learning based scheme for predicting heavy flows directly on the programmable data plane, thus eliminating network overhead and latency to SDN controller. Considering the scarce memory and limited computation capability in the programmable data plane, *pHeavy* includes a packet processing pipeline which deploys pre-trained decision tree models for in-network prediction. We have implemented *pHeavy* in both bmv2 software switch and P4 hardware switch (i.e., Barefoot Tofino). Evaluation results demonstrate that *pHeavy* has achieved 85% and 98% accuracy after receiving the first 5 and 20 packets of a flow respectively, while being able to reduce the size of decision tree by 5.4x on average. More importantly, *pHeavy* can predict heavy flows at line rate on the P4 hardware switch.**

***Index Terms*—Heavy flow, decision tree, programmable data plane, P4.**

I. Introduction

THERE is always a need for network operators to analyze network traffic for improving network reliability, performance, configuration and security management. Heavy flows, which account for a significant fraction of network traffic [36], have profound impact on networks as they either can cause network congestion or indicate an ongoing DoS attack. Therefore, identifying heavy flows correctly and promptly is of great importance to many applications such as mitigating link congestion [14], scheduling of network bandwidth [22] and exposing network attacks [30].

Existing solutions for identifying heavy flows can be broadly categorized as detection or prediction based. Good heavy flow detectors/predictors need to consider tradeoff among three key metrics: *accuracy*, *timeliness* and *network overhead* [37]. This means that the accuracy needs to be high enough for the predictors to be meaningful. The prediction needs to take place as early as possible to give network administrators sufficient early response time. For example, FlowSeer [16] proved that prediction in early response time can effectively enhance the performance of load balancers. Network overhead also should be minimized to avoid impact on normal network traffic.

Statistics-based detection schemes (e.g., Hashpipe [36], PRECISION [12]) collect statistics for monitored flows. However, having a fixed threshold means they are not flexible enough to cope with the dynamism of network traffic. A long detection time is usually required before counters exceeding the fixed threshold. Another way for detection is to sample only a fraction of packets of target flows. However, since monitoring overhead is inevitable in statistics collection, tradeoff between sample frequency (which affecting *accuracy*) and communication *overhead* is still needed.

In comparison to detection, machine learning can predict heavy flows early with high accuracy using only the first few packets of a flow. This gives extra headroom for network operators to take actions on the heavy flows. Most existing machine learning methods need to leverage the SDN architecture [16], [26]. The data plane collects traffic features and sends them to the controller, which runs machine learning algorithms to predict heavy flows. However, such approaches introduce unnecessary or even heavy communication overhead and network delays due to the additional communication between the data plane and controller, violating *timeliness* and *network overhead* attributes above.

Clearly, achieving all three aforementioned metrics simultaneously is challenging. To overcome this challenge, in this paper, we exploit the data plane programmability and propose

Manuscript received January 24, 2021; revised June 22, 2021; accepted June 24, 2021. Date of publication July 5, 2021; date of current version December 9, 2021. This work has been partially supported by Chinese National Research Fund (NSFC) No. 61772235 and 61872239; Natural Science Foundation of Guangdong Province No. 2020A1515010771; Science and Technology Program of Guangzhou No. 202002030372; The UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; InnovateUK grant 106199-47198; Guangdong Key Lab of AI and Multi-modal Data Processing; BNU-UIC Institute of Artificial Intelligence and Future Networks funded by Beijing Normal University (Zhuhai) and AI-DS Research Hub. The associate editor coordinating the review of this article and approving it for publication was D. Carrera. *(Corresponding author: Lin Cui.)*

Xiaoquan Zhang and Lin Cui are with the Department of Computer Science, Jinan University, Guangzhou 510632, China (e-mail: zhangxiaoquan547@gmail.com; tcuilin@jnu.edu.cn).

Fung Po Tso is with the Department of Computer Science, Loughborough University, Loughborough LE11 3TU, U.K. (e-mail: p.tso@lboro.ac.uk).

Weijia Jia is with the BNU–UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with the Guangdong Key Laboratory of AI and Multi-Modal Data Processing, BNU–HKBU United International College, Zhuhai 519087, China (e-mail: jiawj@uic.edu.cn).

Digital Object Identifier 10.1109/TNSM.2021.3094514

pHeavy, a machine learning approach for predicting heavy flows in the programmable data plane. Our results show that *pHeavy* can predict heavy flows with 85% accuracy upon receiving only the first 5 packets of a flow. Moreover, by predicting entirely in the data plane, *pHeavy* eliminates unnecessary communication overhead and network delays between data plane and controller.

It is challenging to train accurate machine learning models that are simple enough to be implemented in the programmable data plane. First, the number of heavy flows is usually a very small fraction of total flows albeit being accountable for most network packets transferred [26]. Such imbalanced distribution can mislead the construction of machine learning models, producing inaccurate and large-sized models consuming more scarce memory of switches. Second, programmable data planes usually have limited memory and computation power, which limits statistics collection and the deployment of prediction algorithms in the programmable data plane[1] [39].

In short, this paper has made the following contributions:

1) A training algorithm is proposed to eliminate effects of imbalanced distribution to obtain accurate models and reduce the size of decision trees for data plane implementation.
2) Considering limited computing resources in programmable data plane, a packet processing pipeline is designed to track each flow's features and predict heavy flows, and a flow management scheme is also enforced to optimize memory usage.
3) A prototype on P4 hardware switch (Barefoot Tofino) is implemented. Evaluation results show that *pHeavy* can predict heavy flows accurately at line rate.

The rest of paper is organized as follows. Section II gives an overview of related works and explains challenges for *pHeavy* design. Section III presents the system overview of *pHeavy*. Sections IV and V describe the detailed *pHeavy* design, including network traffic data training and predicting entirely in the programmable data plane. Section VI discusses the implementation details of hardware and software switches, Section VII shows the evaluation results and Section VIII concludes the paper.

II. Related Works & Challenges

A. Related Works

Recent research works on identifying heavy flow can be divided into two categories (see Table I): statistics-based detector and ML-based predictor.

Statistics-Based Detector: These schemes are based on flow statistics to identify heavy flows. Sampling is a commonly used method [20], which assumes that heavy flows are more likely to be tracked since they generate most of network traffic. Probability sampling [34] can be used to reduce the complexity of detection. However, in order to achieve high accuracy, the sampling process must collect as much flow information as possible. This could introduce high overhead between the data plane and controller. Another way of detecting heavy flow is to periodically collect information, e.g., Hedera [11] and Helios [21]. Similarly, such periodically pulling of statistics of flows may introduce significant network overhead (e.g., 1 to 20MB signaling overhead in data center network [16]) in order to provide precise accuracy. Other works, namely heavy hitter (e.g., Hashpipe [36], DevoFlow [17] and [13]), set up counters in the data plane to detect heavy flow. They count the number of packets transferred in each monitored flow. Although they are easy to implement in switches, they require a long detection time until the counter exceeds a pre-defined fixed threshold. Overall, methods based on statistics can not detect heavy flows in the early stage since they need to wait until the heavy flow bursts.

ML-Based Predictor: Some works adopt machine learning algorithms to predict heavy flow. They are usually implemented in the controller of SDN networks. Poupart *et al.* [33] discussed predicting heavy flow by three machine learning algorithm: neural networks, Gaussian process regression and online Bayesian Moment Matching. Xiao *et al.* [38] utilized a cost-sensitive model in decision trees to train models with high accuracy in heavy flow prediction. FlowSeer [16] designed a two-phase method. The first phase uses cost-sensitive decision trees and the second phase implements data mining with the Hoeffding tree in the controller. Huang *et al.* [26] proposed a heavy flow decision scheme that consists of two stages. The first stage uses *C*4.5 decision tree [35] and the second stage adopts a more accurate machine learning algorithm APPR [25]. However, by deploying ML-based predictors on the controller, they inevitably introduce signaling overhead and delay between the controller and data plane.

B. Challenges

Two important challenges must be considered in actual design and implementation of an machine learning based predictor for heavy flows. First, imbalanced distribution of heavy flows can lead to an inappropriate model during training. Second, programmable hardware data planes usually have limited memory and computation capability. This means any resulting models will need to be lightweight for them to be run on the hardware data plane.

1) Imbalanced Data Problem in Heavy Flow Prediction: In practice, network traffic contains about 10% of heavy flows. Interestingly they carry more than 90% of total packets [37]. *Such extreme imbalanced distribution will mislead the classifier, causing unacceptable accuracy*. This is called imbalanced data problem in machine learning [24]. Since most algorithms assume balanced class distributions or equal misclassification costs, these algorithms will fail to represent the distributive characteristics of the data and result in low prediction accuracy across the classes of imbalanced data. For example, *C*4.5 decision tree uses IGR (Information Gain Ratio) to select decision features and determine branches, and features with high IGR can be used to discriminate different classes well. However, this mechanism does not consider the accuracy of classification of the minority class. Furthermore, these solutions may lead to large-sized models (Experiments in Section IV-C), which

[1]For example, Barefoot Tofino [4] only supports 12 stages in the pipeline and dozens of megabytes of available memory, and does not support multiplication and division.

TABLE I
COMPARISON OF *pHeavy* WITH OTHER SOLUTIONS

Type	ML-based		Statistics-based	
Location	Data plane	Controller	Data plane	Controller
Method	Decision tree	Decision tree & Hoeffding tree	Heavy hitter	Simple & probability sampling
Examples	*pHeavy*	FlowSeer [16] Huang et al. [26] Xiao et al. [38]	Basat et al. [13] DevoFlow [17] Hashpipe [36]	Netflow [20] SIFT [34]
Detection/prediction delay	Low (e.g., 90% of flows within 1.2 seconds in UNI1 dataset)	Prediction & communication delay	High (e.g., 90.3% of flows within 2.0 seconds in UNI1 dataset)	High
Switch-controller communication overhead and delays	None	Moderate	None	Heavy
Accuracy	High (e.g., 20 packets with 97.6% TNR in UNI2 dataset)	High	Low (e.g., 20 packets with 79.5% TNR in UNI2 dataset)	Low

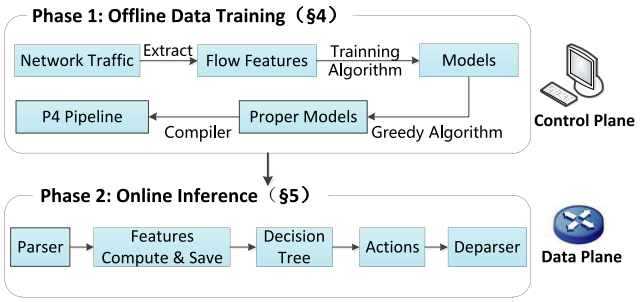


Fig. 1. The overview of *pHeavy* .

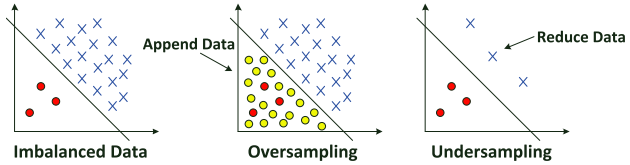


Fig. 2. Two sampling methods. Oversampling appends data to be balanced dataset and undersampling reduces data to be balanced dataset.

can not be implemented in the programmable data plane with limited memory (e.g., TCAM).

2) Offloading in the Data Plane: Programmable switches usually have limited memory and computation power due to the cost of high-speed hardware. This creates challenges for the implementation of *pHeavy*. Scarce memory (e.g., few tens of megabytes) in a switch will limit the size of machine learning models and the number of flows it can monitor. Some optimization techniques, such as removing inactive flows and their associated flow status from the flow tables, could be used to free up more memory, but programmable data planes currently lack dynamic memory management to support such optimization. Furthermore, most machine learning algorithms (e.g., SVM or neural network) cannot be implemented in the programmable data plane due to the requirement of floating operations. Moreover, some frequently-used statistics operations such as multiplication/division and average are also difficult to implement.

III. SYSTEM OVERVIEW

This section provides an overview of *pHeavy* which consists of an offline model training and online inference as illustrated in Figure 1.

Offline Model Training: In this phase, the controller trains machine learning models and compiles them into components of target switches which enables prediction at runtime in the second phase. *pHeavy* uses real network datasets [1] containing network information in the transport and network layers (e.g., in PCAP format) as the input in the first phase, extracts features of each flow from the dataset and labels each flow in advance.

Afterwards, *pHeavy* utilizes its training algorithm to train models, in which the undersampling method randomly generates different training datasets that can train models with different performance. It provides more options to achieve a tradeoff between the size of models and accuracy. Since undersampling will produce multiple trained models with different performance, a greedy algorithm is also designed to determine appropriate models with better performance that can be deployed to the data plane. Lastly, the selected decision trees will be compiled for target switches.

Online Inference: The second phase is predicting heavy flow in the programmable data plane. *pHeavy* uses decision trees because of their simplicity in components (e.g., conditional statements) and operations (e.g., does not need floating), which can be easily implemented in the programmable data plane. The *pHeavy* pipeline firstly extracts packet's header by the parser module for computing flow features, and then saves them in registers. When the number of received packets of a flow reaches a preset value, e.g., 5 packets, the flow will be verified by a pre-installed decision tree and tagged with a flag (e.g., heavy or non-heavy flow). After a flow is judged as a certain type, its subsequent packets will not modify its features and network operators can apply their desired actions based on the requirements of network applications.

IV. NETWORK TRAFFIC TRAINING

This section focuses on how to train machine learning models for *pHeavy*.

A. Heavy Flow and Metrics

Heavy Flow: A flow f_i is a sequence of packets with the same 5-tuple (source IP, destination IP, source port, destination

TABLE II
THE CONFUSION MATRIX OF BINARY CLASSIFICATION

	1	0
1	True Positive (TP)	False Negative (FN)
0	False Positive (FP)	True Negative (TN)

port, protocol). M_i is denoted as the total length of packets of f_i. A subflow is denoted as $f(i:j)$ and subsequently the first j packets of a flow are denoted by $f(1:j)$.

Definition 1: Given a data stream consisting of a set of flows F, a flow $f_i \in F$ is defined as a *heavy flow* with respect to a customized threshold ϕ:

$$M_i \geq \phi$$

And, the *occupation rate* of heavy flow in F is defined as:

$$\alpha = \frac{|\{f_i | M_i \geq \phi, f_i \in F\}|}{|F|} \tag{1}$$

Confusion Matrix and Metrics: Identifying heavy flows can be seen as a binary classification, where the 0 class represents non-heavy flow and 1 class is heavy flow. Given the two-class case, a confusion matrix is shown in Table II.

True Positive (TP) means the observation is positive, and the sample is predicted to be positive. False Negative (FN) is that the observation is positive, but the sample is predicted to be negative. True Negative (TN) describes that observation is negative, and the sample is predicted to be negative. And False Positive (FP) represents that observation is negative, but the sample is predicted to be positive.

Two metrics are used to evaluate the accuracy: true positive rate (TPR) and true negative rate (TNR) [33]. TPR is the ratio of the total number of correctly classified positive samples to the total number of positive samples, and TNR is the ratio of the total number of correctly classified negative samples to the total number of negative samples:

$$TPR = \frac{TP}{TP + FN}$$
$$TNR = \frac{TN}{TN + FP}$$

Thus, TPR and TNR represent the percentage of heavy flows that are correctly predicted and the percentage of non-heavy flows that are correctly predicted, respectively [33]. Therefore, *the objective of pHeavy is to achieve high TPR and TNR.*

B. Network Traffic Features

There are many useful network traffic features revealed from previous works [32]. Considering limitations of the programmable data plane, *pHeavy* only selects features that are feasible to be implemented in the programmable data plane. There are two types of features: stateless and stateful features, as shown in Table III. Stateless features refer to intrinsic characteristics of a flow (e.g., destination TCP/UDP port). And stateful features should be saved and computed in the programmable data plane (e.g., total length).

TABLE III
NETWORK TRAFFIC FEATURES

Name	Type	Description
IAT_min	Stateful	Minimum of packet inter-arrival time
IAT_max	Stateful	Maximum of packet inter-arrival time
IAT_avg	Stateful	Average of packet inter-arrival time
IAT_total	Stateful	Total of packet inter-arrival time
len_min	Stateful	Minimum of packet length
len_max	Stateful	Maximum of packet length
len_avg	Stateful	Average of packet length
len_total	Stateful	Total of packet length
SYN/ACK	Stateful	TCP SYN/ACK flag counter
PSH/ECE/RST	Stateful	TCP PSH/ECE/RST flag counter
sport/dport	Stateless	Source/Destination port

C. Imbalanced Data Problem

Imbalanced Data Problem Model: Assuming an imbalanced distribution consists of a minority class and a majority class. Considering a given dataset S, two subsets $S_{min} \subset S$ and $S_{maj} \subset S$ are defined. S_{min} is the set of minority samples in S and S_{maj} is the set of majority samples in S. And, $|S_{maj}| \gg |S_{min}|$ and $|S| = |S_{maj}| + |S_{min}|$.

Cost-Effective Learning and Sampling: The basic idea of cost-sensitive learning is the concept of cost matrix. Cost matrix numerically represents the penalty of classifying samples from one class to another. Assuming a binary classification, the cost matrix can be defined as:

$$C(i,j) = \begin{pmatrix} 0 & c \\ d & 0 \end{pmatrix}$$

where d is denoted as the cost of misclassifying a majority class (in heavy flow problem, setting d to 1 represents no penalty for false negative), and c is denoted as the cost of misclassifying a minority class. The objective of cost-sensitive learning is to minimize overall cost on training data, and it changes the class i of each flow f to the class j such that $\sum_i P(i|\boldsymbol{S})C(i,j)$, where $P(i|\boldsymbol{S})$ represents the probability of each class i for a given training dataset $\boldsymbol{S}$. Subsequently, tuning a cost matrix value can obtain a favorable model for training data.

Sampling methods aim to balance the number of minority class and majority class. Undersampling removes samples of the majority class (e.g., random-based [3]). So,

$$|S_{new}| = |S_{min}| + |S_{maj}| - |S_{under}| \tag{2}$$

where S_{new} is defined as the new data set after sampling and $S_{under} \approx S_{maj} - S_{min}$ is the data set that is reduced from the majority data set. Oversampling appends samples to the minority class (e.g., clustering-based [24]). So,

$$|S_{new}| = |S_{min}| + |S_{maj}| + |S_{over}| \tag{3}$$

where $S_{over} \approx S_{maj} - S_{min}$ is the data set that is appended to minority data set. Figure 2 illustrates difference of the two solutions. The Equation (3) implies that the new constructed dataset will produce more complicated machine learning models, which is difficult to be executed in the data plane. In

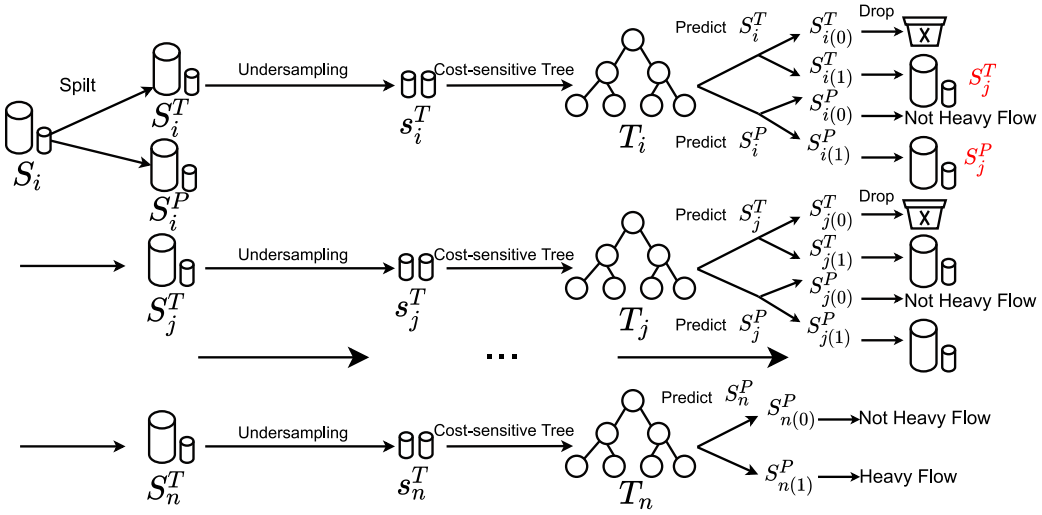


Fig. 3. The training algorithm consists of several stages, each of which generates a decision tree, a training dataset and a predicted dataset for next stage (marked by red), until the final stage determines heavy flow.

TABLE IV
PERFORMANCE ANALYSIS ON IMBALANCED DATA (METRICS ARE INTRODUCED IN SECTIONS IV-A AND VII-A)

10% of heavy flow in UNI2			
	Cost-sensitive	Undersampling	Oversampling
TPR	**0.834**	**0.854**	0.663
TNR	0.760	0.760	0.881
F-measure	0.537	0.546	0.567
Tree size	969	**273**	1083
2% of heavy flow in UNI2			
	Cost-sensitive	Undersampling	Oversampling
TPR	0.379	**0.838**	0.415
TNR	0.958	0.781	0.960
F-measure	0.287	0.180	0.295
Tree size	979	**81**	1011

contrast, the reduced amount of the dataset in Equation (2) enables more simple machine learning models.

Data Trace Driven Analysis: To investigate the performance of above methods, a trace driven analysis is conducted using a real network dataset (i.e., UNI2 [1]). Information of each flow in the dataset is analyzed and more than ten common network features are computed based on the first 20 packets of each flow. The cost-sensitive learning and two sampling methods are used to predict heavy flows under two different occupation rates, i.e., 2% and 10% [16].

Three common machine learning metrics and the size of decision trees are used to evaluate, as shown in Table IV. Results show that only undersampling method has the highest TPR with the smallest tree size. However, its tree size is still too large to be implemented in programmable data plane. Furthermore, when the occupation rate of heavy flows is reduced from 10% to 2%, both TPR and F-measure of all methods are decreased, which indicates that neither of these methods can independently handle imbalanced distribution of heavy flow effectively. Notice that the cost-sensitive method can obtain high TPR. Thus, *pHeavy* adopts a training scheme combining the undersampling and cost-sensitive (e.g., metacost [18]) methods as the fundamental algorithm to overcome the imbalanced data problem. Experiments show that, after going through several decision trees built by different undersampling training data, *pHeavy* can achieve the same accuracy as the method based on controller.

D. Decision Tree Training

The training algorithm of *pHeavy* is depicted in Figure 3. S_i is defined as dataset consisting of features of subflows ($f(1:i)$), and S_i^T and S_i^P are defined as original training dataset and verifying dataset respectively splitted from S_i. s_i^T is subset of S_i^T after sampling. T_i is defined as a decision tree that predicts heavy flows after receiving the i^{th} packet. Four datasets are defined to represent results of the decision tree T_i with tagging class 0 or class 1: $S_i^T{}_{(0)}$, $S_i^T{}_{(1)}$, $S_i^P{}_{(0)}$, and $S_i^P{}_{(1)}$, where $S_i^T{}_{(0)}$ and $S_i^T{}_{(1)}$ are prediction results of S_i^T, and $S_i^P{}_{(0)}$ and $S_i^P{}_{(1)}$ are prediction results of S_i^P.

The whole training process consists of multiple stages as shown in Figure 3. Each stage includes several steps. Initially, original dataset is divided into original training dataset and verifying dataset. The former would be reduced to the balanced training dataset (e.g., s_i^T) through the two undersampling methods, i.e., OSS (One-Sided Selection) [29] and Random [3]. The OSS selects a representative subset of the majority class to combine it with the minority class as a new training dataset, which removes samples that have similar characteristics with the sample of the majority class to compact the majority class. The Random undersampling equitably balances the amount of the majority class samples and the minority class samples. Each randomly selected sample can only represent parts of characteristics of the majority class, so that it generates simple models with low TNR. Leveraging the property of Random undersampling, *pHeavy* lets each flow go through one or several simple machine learning model(s) for high TPR and TNR. Then, the balanced training dataset is used to train a decision tree (e.g., T_i) with high TPR and low TNR by the cost-sensitive algorithm. In other words, the decision tree aims to filter non-heavy flows but maintain as many

Algorithm 1 Greedy Searching Algorithm

Input: Q, the number of tree
thr, threshold of tree score
$S^T = \{S_i^T, S_j^T, \ldots\}$, training dataset
i, n, define the starting and stopping search location
E, the searching pace
Output: L, list of result consisting of decision trees

$Starter \leftarrow i + 1$
$m \leftarrow 0$ //initialize number of tree
$L \leftarrow \emptyset$ // initialize list of result
while $m < Q$ && $Starter < n$ **do**
  $m \leftarrow m + 1$
  $j \leftarrow Starter$ // move *j* to search
  $T\_list \leftarrow \emptyset$ // save training result
  // - - - search satisfied tree
  while $score\_temp < thr$ **do**
    $j \leftarrow j + 1$
    $T\_list \leftarrow$ Train(S_j^T) // training process
    $(T\_temp, score\_temp) \leftarrow$ Max($T\_list.score$)
  end while
  // - - - save satisfied tree
  L.insert(0, T_temp)
  $Starter \leftarrow j$
  // - - - expand searching range
  $j_e \leftarrow E$
  while $j_e > 0$ **do**
    $j_e \leftarrow j_e - 1$
    $j \leftarrow j + 1$
    $T\_list \leftarrow$ Train(S_j^T)
    $(T\_temp, score\_e) \leftarrow$ Max($T\_list.score$)
    if $score\_e > score\_temp$ **then**
      L.pop(0)
      L.insert(0, T_temp)
      $Starter \leftarrow j$
    end if
  end while
end while

heavy flows as possible. Next, in each prediction step, the decision tree will give two predicted results: class 0 (non-heavy flow) or class 1 (heavy flow). The dataset with class 0 will be classified as non-heavy flow, and the dataset with class 1 will become a new prediction dataset (e.g., S_j^T) for the next stage. To increase accuracy of subsequent training models, *pHeavy* also predicts the training data (e.g., S_i^P) and utilizes the results as the new training dataset (e.g., S_j^P) in next stage. It will delete the flows whose characteristic has been integrated in the decision tree, so that the Random undersampling method would not select them in the next stage.

E. Selecting Decision Trees

The Random undersampling method will produce multiple datasets to train decision trees with different performance. To obtain better performance with limited resource consumption in the data plane, *pHeavy* adopts a greedy searching algorithm (see Algorithm 1) to select several optimized decision trees in certain locations (e.g., i^{th} packet).

Criterion for Tree Selection: pHeavy selects a decision tree based on three conditions: small size, high TPR and high TNR. A score is calculated based on these three conditions for all decision trees. Then, a tree will be selected if its score exceeds a predefined threshold.

Greedy Searching Algorithm: A greedy algorithm is designed to select specific number (e.g., *Q*) of decision trees satisfying the criterion (e.g., *thr*) in a searching range (e.g., *i* to *n*) as early as possible. Our primary experimental analysis has shown that small interval between predicting locations of two decision trees may decrease the performance of the later trees. To address the problem, a variable *E* is used as the pace to expand searching range to improve performance when a satisfied tree is found.

To show the effectiveness of the greedy algorithm, we also implement a random method for decision tree selection, which will randomly select four predicting locations and each location has a trained decision tree.

V. Prediction in Data Plane

This section explains how online inference works and addresses limitations in the data plane of P4 switch.

A. Online Inference in the Pipeline

The processing pipeline of online inference in the programmable data plane is illustrated in Figure 4.

Parser: The parser component parses packet headers which are used to extract and compute features, e.g., TCP flags. This information will traverse with the packet to following stages.

Ingress & Egress: A packet will be firstly hashed to a corresponding flow and then be processed differently: (*i*) if the packet is attached with termination TCP flags or it exceeds IAT limit, its corresponding flow's memory space will be initialized (e.g., set to 0); (*ii*) if the flow has been categorized as a heavy flow, predefined actions (e.g., driving flows to leisure links for load balancing [16]) will be applied; (*iii*) the update of corresponding flow's features and prediction by a decision tree (e.g., T_5) will be triggered when it satisfies predefined conditions (e.g., the arrival of the 5^{th} packet). In the end of the pipeline, all packets need to be deparsed for packet construction before their departure.

Computing Components: pHeavy uses 5 tuples of a packet, i.e., {source IP address, destination IP address, source port, destination port, protocol}, to uniquely identify a flow by hash algorithms which P4 supports. In addition, P4 switch offers timestamps when the packet enters the ingress pipeline, which can be used to compute the time related variables (e.g., IAT).

All features are stored in registers and computed by basic operations supported by P4 (e.g., addition, subtraction, hash). For example, the value of ACK flag counter will be increased if the ACK flag of an incoming packet is set. Since P4 does not support division operation, *pHeavy* uses exponentially weighted moving average (EWMA) to implement average

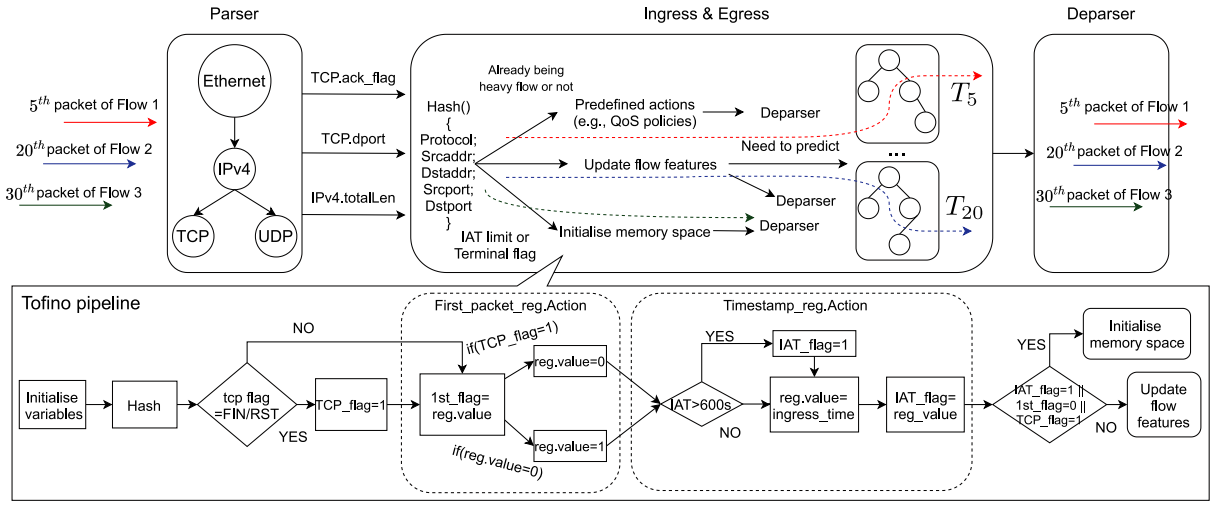


Fig. 4. *pHeavy* 's pipeline and the process of flow management in the P4 hardware switch.

operation [15]. The EWMA can be represented as:

$$S_t = \begin{cases} y_1, & t = 1 \\ \alpha \cdot y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

Let $\alpha = 0.5$, which can be implemented through bit shift.

B. Compilation of Decision Tree in the P4 Switch

P4 allows metadata traversing through different stages in the pipeline. Once prediction is triggered, *pHeavy* fetches the values of features from registers and saves them in metadata as inputs of the decision trees in following stages.

There are two ways to realize decision trees in the data plane: (i) Decision trees can be implemented in the control flow block using variables and the conditional statement (i.e., *if* and *else*). Variables store the value of nodes of the decision tree, and the function of conditional statements resemble branches. For example, assuming a node *dport* $>$ 1234, the variable saves the value 1234 and pHeavy uses *if* and *else* to control different executions. (ii) Each root node of a decision tree is a result of prediction, and it is destined by all features of the decision tree which have specific value ranges. For example, the prediction result of a node is a heavy flow, and the values of all features satisfying the result are $A \in [a_1, a_2]$ and $B \in [b_1, b_2]$, etc. This process is similar to the flow table in P4 switch using range matching. Each entry in the table has match fields and actions. The match fields can be represented as features' value range and the actions will set the prediction result of flows (e.g., flag). Thus, a decision tree can be transferred into a table with different entries in the pipeline. The larger the decision tree, the more table entries are required, thereby consuming more memory (e.g., TCAM).

A flow needs to go through one or several decision trees in order to reach an identification (e.g., $T_i, \ldots, T_n$). In other words, except for the final decision tree, all intermediate decision trees do not identify any heavy flows, but can make decisions on non-heavy flows. Hence, *pHeavy* uses three flags to tag it. Flag 0 means a flow is undetermined and needs continuous monitoring. Flag 1 and flag 2 represent non-heavy flow and heavy flow respectively. Figure 4 gives an example of three distinct flows that traverse the pipeline. For each flow, the verification of each decision tree happens only once when the switch receives the i^{th} packet of the flow. For example, the 5^{th} packet of flow 1 (red dashed line) and the 20^{th} packet of flow 2 (blue dashed line) are predicted by T_5 and T_{20} respectively.

C. Memory Management

Since the number of registers is much smaller than the number of flows, *pHeavy* adopts a memory management strategy to allocate memory dynamically.

There are two rules indicating termination or expiration of a flow. Considering a TCP flow sending signal packets to inform connection termination (e.g., FIN flag and RST flag), *pHeavy* exploits these TCP flags as the sign of termination of a flow. Upon receiving a packet with such termination flags, a flag of its corresponding row space in the register of the flow will be set, indicating the register is released. On the other hand, real network trace [1] shows that some flows only transfer few packets without any termination TCP flags. Therefore, *pHeavy* uses interval arrival timeout as another signal of flow termination (i.e., *pHeavy* sets IAT>600 seconds). Due to the lack of packets with termination flags, an expiration flag can only be tagged by other flows which hash to the same slot. Once hash collision happens and the register space is full, a new flow can reuse the memory space of the original flow that has been expired or terminated.

VI. Implementation

We have implemented *pHeavy* on bmv2 [5] consisting of over 700 lines of $P4_{16}$ code (including several decision trees with maximum depth 10), and P4 hardware switch (Flnet S9180-32X) with a 3.2Tb/s Barefoot Tofino 32D ASIC [4].

A. Offline Model Training

UNIBS Dataset: The UNIBS dataset [9], [19], [23] were collected on the edge router of the campus network of Brescia University during three consecutive working days. We use the

day3 traffic trace which is mainly composed of TCP (99%) and UDP to evaluate *pHeavy*.

UNI Dataset: The UNI dataset [1] has two packet traces from two university data centers, UNI1 and UNI2. The major traffic in UNI1 is TCP traffic. By contrast, most heavy flows are UDP traffic in UNI2. Although the number of TCP flags would not be useful in the UDP flow, experiments in UNI2 show that *pHeavy* also performs well in UDP flows.

Model Training: The model training is completed in Weka 3 [10]. *pHeavy* uses dpkt [2] to extract features of network traffic and RandomUnderSampler [3] to mitigate imbalanced data problem. scikit-learn [7] is used to split dataset into training dataset (70%) and testing dataset (30%) for the method "holdouts" validation [27], [31].

Threshold: The value of a threshold will affect the number of flows that are recognized as heavy flows, and hence impacts the accuracy of prediction. However, there is no unanimous value for the threshold. For example, Helios [21] defined heavy flow with flow rate below 15Mbps, and Devoflow [17] defined three thresholds with different values (128KB, 1MB and 10MB). To satisfy the needs of different networking environments, *pHeavy* allows network operators setting the threshold, e.g., occupation rate α (Section IV-A), for heavy flow according to their requirements.

B. pHeavy in the P4 Hardware Switch

Following aspects are considered when implementing *pHeavy* on a Barefoot Tofino P4 hardware switch.

Register Operation: In P4 hardware switches, all operations on the same *Register* must be in the same stage. For example, in the update of a *Register* used to save the feature value, it may require several operations: reading the value, calculation and writing a new value. *pHeavy* adopts the *RegisterAction* block provided by $P4_{16(Tofino)}$ to aggregate all operations for a *Register* in the same stage.

Features: Since Barefoot Tofino only offers limited operations, *pHeavy* only uses a small part of flow features, as shown in Table III. In the experiment, it was found that the prediction has close accuracy with prediction in the software switch and the size of decision trees is only marginally increased (e.g., 5% in UNIBS dataset). Table V lists the top three important features in each decision tree based on information gain ratio [35], showing which feature is more important in the prediction. In dataset UNI1, TCP flags are important features in software switches (bmv2) and P4 hardware switches (Tofino). In comparison, UNI2 prefers length and IAT features since the dataset mainly consists of UDP traffic.

Memory Management in P4 Hardware Switch: There are two major challenges for implementing memory management in Barefoot Tofino: (i) Due to the limited number of stages (e.g., 12 stages) in the pipeline, *pHeavy* needs to use as few stages as possible to save space for other applications. (ii) Although *RegisterAction* allows programmers to access *Register* multiple times, the *RegisterAction* block can not be split into sub-blocks for using at different stages.

To overcome the two challenges, *pHeavy* defines two *Registers* and three flag variables. The two *Registers*, i.e., *First_packet_reg* and *Timestamp_reg* are shown in the two dotted boxes in Figure 4. *First_packet_reg* is used to record whether the packet is the first packet of a flow, and *Timestamp_reg* records the last packet's arrival time of a flow. In the meantime, three flag variables, i.e., *IAT_flag*, *1st_flag* and *TCP_flag*, are used to determine whether the memory space need to be initialized by three conditions: (i) exceeding IAT threshold, (ii) arrival of the first packet of a flow and (iii) receiving TCP termination flags.

TABLE V
TOP THREE IMPORTANT FEATURES IN DECISION TREES

UNI1			UNI2	
# of packets	bmv2	Tofino (TCP)	# of packets	bmv2
6th	IAT_max dport IAT_total	PSH SYN len_max	5th	dport IAT_max len_min
8th/9th	len_avg dport ACK	ACK len_total SYN	7th	len_max len_avg len_total
14th	SYN dport ACK	ACK dport SYN	14th	IAT_total len_min IAT_avg
20th	ACK PSH len_avg	dport ACK	20th	IAT_avg IAT_total len_max

The process of memory management in the pipeline is shown in Figure 4. A packet attached with TCP termination flags can trigger memory initialization, setting the variable *TCP_flag* to 1. If it directly initializes memory space, the termination packet will be the first packet of a new flow hashed in the same memory space, i.e., the next packet will be the real first packet. Thus, *pHeavy* uses *First_packet_reg* to record whether the packet is the first packet after processing TCP termination packets. In the *RegisterAction* of *First_packet_reg*, the TCP termination packet (*TCP_flag* $= 1$) will set the value of the *Register* to 0. Then the next packet (i.e., the first packet of a flow) fetches the value of *First_packet_reg* (now equals 0) to initialize memory space while setting the value of *First_packet_reg* to 1 for updating subsequent packets. Afterwards, in the *RegisterAction* of *Timestamp_reg*, a packet's arrival time (e.g., ingress timestamp) will be used to judge whether it triggers memory initialization. If the IAT threshold has been exceeded, *IAT_flag* will be set to 1 while updating the value of *Register*, and the packet is the first packet of a new flow on this condition. Otherwise, only update the value of *Register*. In the end of the process, values of these three flags will determine whether to perform feature updating or memory initializing operations.

VII. EVALUATION

In this section, we evaluate the performance of *pHeavy* in both software switch (i.e., bmv2) and P4 hardware switch (i.e., Barefoot Tofino).

A. Performance Metric and Comparison Schemes

F-Measure: Since the change of occupation rate may not effect TPR and TNR, these two metrics can not correctly reflect the performance of classifiers in imbalanced dataset.

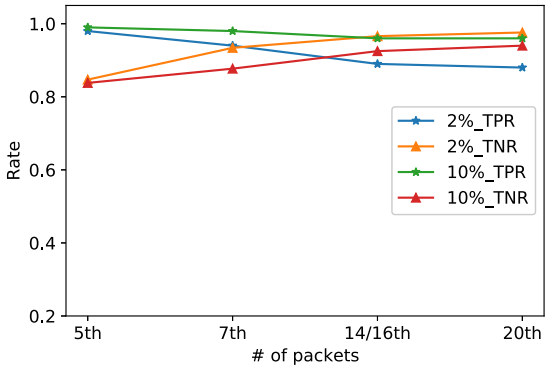


Fig. 5. *pHeavy* selects four decision trees in different locations for prediction when $\alpha = 10\%$ and $\alpha = 2\%$ respectively.

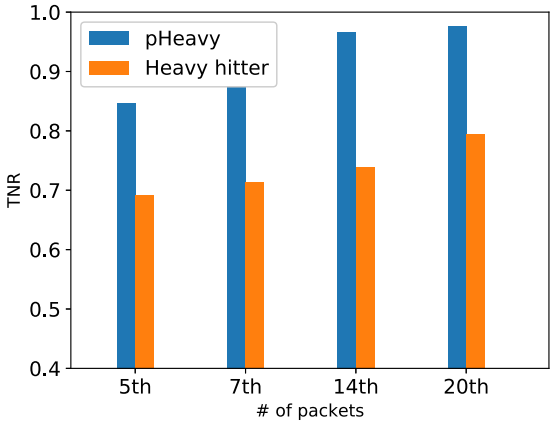


Fig. 6. Comparison of TNR with heavy hitter.

F-measure ($\beta = 1$) is a metric widely used to evaluate binary classification. F-measure metric combines precision and recall as an effectiveness measure of classification, defined as:

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

F-measure has more insights into the functionality of a classifier. It is very sensitive to the rate of imbalance of dataset [28]. The value of F-measure will be attenuated severely by more skewed distributions. Therefore, to evaluate the performance of *pHeavy* in extremely imbalanced distribution, F-measure can be used to evaluate the same dataset with two occupation rates of heavy flow ($\alpha = 2\%$ and $\alpha = 10\%$).

Heavy Hitter: Heavy hitter [36] is a common method based on counters for heavy flow detection in the data plane. It can be easily integrated into switches but has a long detection time. Thus, we compare the TNR of *pHeavy* and heavy hitter after receiving few packets of each flow.

Machine Learning in the Controller: APPR [25] is a machine learning scheme for predicting network traffic. Previous work has run APPR algorithm in the controller to identify heavy flows [26] since it has high accuracy of prediction by only few packets of a flow. Considering the imbalanced data problem, random undersampling is performed on training data to keep the same number of minority and majority when training models by APPR.

B. Performance in P4 Software Switch

Accuracy: Figure 5 shows the training performance of *pHeavy* on both TPR and TNR. The experiment is conducted on UNI2 dataset. *pHeavy* arranges four decision trees for prediction in different locations, i.e., the prediction happens in the 5/7/14/20*th* packet when $\alpha = 2\%$ and the 5/7/16/20*th* packet when $\alpha = 10\%$ respectively. The result shows that *pHeavy* can keep high TPR while obtain high TNR. TPR decreases while TNR increases when a flow receives more packets. The reason is that each decision tree aims to keep high TPR while increasing TNR as high as possible. The first decision tree has a high TNR, partially contributed by correctly identifying flows not exceeding four packets, and these flows can be erased by the memory management subsequently.

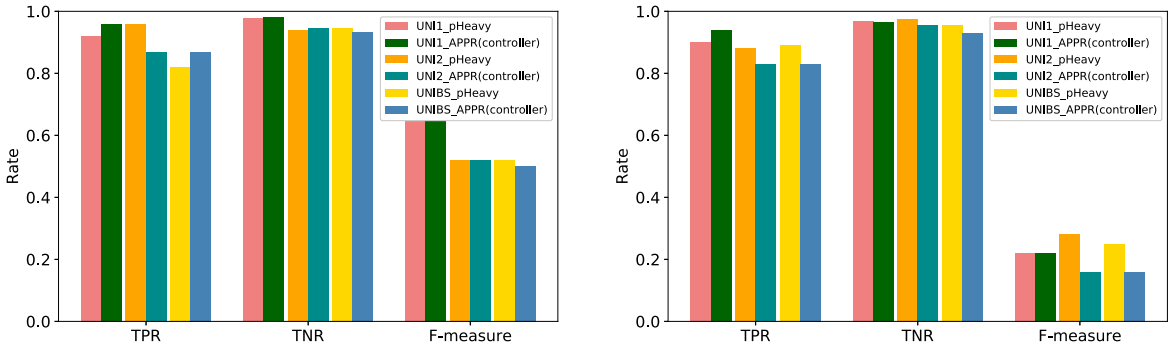
Comparison With Heavy Hitter: Figure 6 shows the speed of prediction for *pHeavy* and heavy hitter. As we can see from this figure, heavy hitter is able to achieve 70% TNR because of the existence of short-lived flows. The TNR for both schemes grows steadily as more packets enter the switch, but our *pHeavy* consistently has more than 15% better performance.

Comparison With Machine Learning in Controller: We next compare the performance of *pHeavy* and APPR. This experiment evaluates three metrics, including TPR, TNR and F-measure, and is conducted on the three datasets (UNI1, UNI2 and UNIBS) with two occupation rates ($\alpha = 10\%$ and $\alpha = 2\%$). Results in Figure 7 shows that *pHeavy* has comparable performance with APPR. First, both *pHeavy* and the APPR algorithm have high TPR and TNR. Second, *pHeavy* has good performance in F-measure when comparing with APPR. The value of F-measure is attenuated in *pHeavy* and APPR when the occupation rate of heavy flow declines. It is worth noting that *pHeavy* has a higher F-measure value when $\alpha = 2\%$. This is because the flow memory management of *pHeavy* filters out a portion of flows that meet the termination conditions before triggering the prediction of the decision tree. However, the core idea of APPR is defining "application layer round", which largely extends the number of available features to increase accuracy. Thus, APPR is inappropriate to be implemented in the data plane as it requires so many features and produces complicated machine learning models.

C. Performance in the P4 Hardware Switch

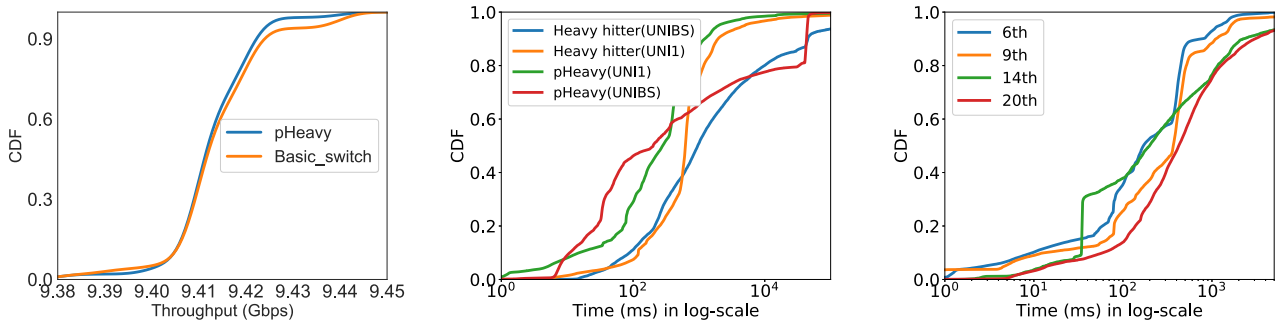
Our testbed for evaluating *pHeavy*'s performance in P4 hardware switch consists of two servers (equipped with 8 Intel Core i7-4771 CPU @ 3.50GHz), each configured with a 10Gbps NIC, and a P4 hardware switch (Flnet S9180-32X with Barefoot Tofino) connecting the servers. One server is used to replay real TCP network traffic (i.e., UNI1 and UNIBS) via Linux network traffic tool *Tcpreplay* [8], and the other one is responsible to receive the replayed packets.

Predicting at Line Rate: To evaluate the throughput of *pHeavy*, a simple switching application named *basic_switch* [4] is implemented for baseline comparison. *iperf* is used to measure throughput between the two servers. As we can see in Figure 8(a), *pHeavy* achieves an average throughput of 9.412Gbps, which is only 0.01% slower than 9.413Gbps



(a) 10% of heavy flow (b) 2% of heavy flow

Fig. 7. Comparison of *pHeavy* and APPR algorithm by three metrics.



(a) Throughput of *pHeavy* and a simple use case in P4 hardware switches (b) Comparison of pHeavy and heavy hitter in flow predicting time (c) Flow prediction time at different locations (UNI1)

Fig. 8. Performance on the P4 hardware switch.

achieved by the *basic_switch*. This clearly demonstrates that *pHeavy* is able to work at line rate.

Flow Prediction Time: Flow prediction time refers to the time interval between receiving the first packet of a flow to predicting it as a heavy flow or a non-heavy flow. Flows that do not trigger prediction (e.g., the number of packet is less than 5) are not included. We also implement heavy hitter which maintains a counter for each flow and set its threshold to the number that is the same as the position of the last decision tree (i.e., 20^{th}) in *pHeavy* for comparison. Figure 8(b) shows the CDF of flow predicting time of both *pHeavy* and heavy hitter with UNI1 and UNIBS datasets. Evaluation results show that about 90% of flows can be predicted within 1.2s in UNI1 dataset. And the average flow prediction time is 2.6s and 10.3s for UNI1 and UNIBS respectively. In comparison, the time for heavy hitter is 7.9s and 31.8s respectively. Thus, *pHeavy*'s prediction time is 3x faster than heavy hitter on average. In the UNIBS dataset, flow predicting time of *pHeavy* increases because the dataset consists of many SSH tunnel flows that only transfer dozens of packets over tens of seconds. It is noted that heavy hitter has very low TNR (e.g., about 80% TNR in 20^{th} packets, as shown in Figure 6) when it has the same predicting location with *pHeavy*.

In addition, Figure 8(c) shows flow predicting time in different predicting locations (i.e., 6^{th}, 9^{th}, 14^{th} and 20^{th}). As expected, the earlier the location of prediction, the shorter the flow predicting time. Flow predicting times of over 91% at the 6^{th} and 9^{th} packets, 84.7% at the 14^{th} packets, and 75.4% at the 20^{th} packets are completed within 1s.

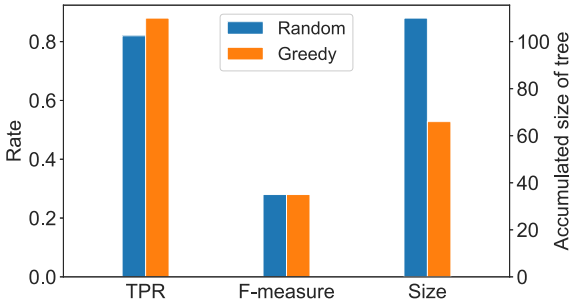


Fig. 9. Comparison of random algorithm and greedy algorithm for decision tree selection.

D. Optimization for the Programmable Data Plane

Selection of Decision Trees: pHeavy adopts a greedy algorithm to select decision trees. To show the effectiveness of the selection algorithm, a random method is also designed for comparison. As shown in Figure 9, although the two methods have the same F-measure value, the greedy algorithm has a higher TPR while smaller accumulated size of decision trees. Thus, the greedy algorithm can effectively select decision trees to be implemented in the data plane.

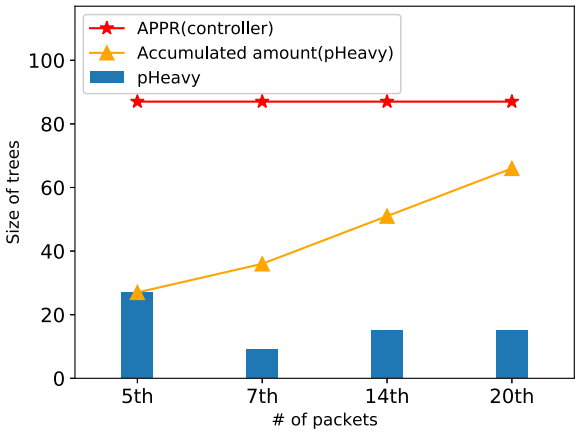


Fig. 10. Comparison of the size of decision trees with APPR algorithm.

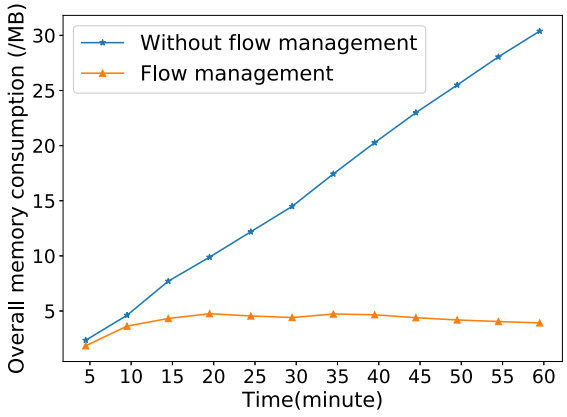


Fig. 11. The overall memory consumption in different time conducted by UNI2 dataset.

Optimization on the Size of Decision Trees: Figure 10 shows the performance of the training algorithm for both *pHeavy* and APPR in terms of the size of decision trees (e.g., amounts of nodes). Decision trees produced by *pHeavy* are smaller by 5.4x on average than that of APPR. Although the accumulated size in *pHeavy* is close to that of APPR, *pHeavy* shares computation of prediction in a flow with several smaller predictions in different packet locations, which effectively enables it to avoid effects of processing per packet in the data plane. Thus, *pHeavy* improves the feasibility of implementing decision trees in the programmable data plane.

Optimization on the Amount of Stored Flows: Since the scarce memory in the programmable data plane limits the storage of flows, we also study the number of flows that *pHeavy* can concurrently classify by giving a certain memory. Similar to [15], the number of concurrent flows is estimated according to the given memory divided by total bits of information to be stored per-flow. Storing all mentioned features of each flow, *pHeavy* can handle about 200 thousands of concurrent flows per 10MB memory. Furthermore, Figure 11 shows overall memory consumption that *pHeavy* needs to provide in different time spots during experiments. *pHeavy* only requires 5MB memory in the P4 software switch to store all features of up to 100k active flows. Thus, *pHeavy* is able to schedule the memory usage effectively and dynamically.

VIII. Conclusion

In this paper, we present *pHeavy* which predicts heavy flow via machine learning algorithm in the programmable data plane. *pHeavy* consists of two phases, offline model training and online inference. In the first phase, *pHeavy* proposes a training algorithm to tackle the imbalanced data problem while minimizes the size of trees to implement in the data plane. In the second phase, *pHeavy* provides memory management to increase the amount of concurrent flow, and replaces unsupported operations with approximate values. Experiment results show that *pHeavy* can effectively predict heavy flow in early stages with high accuracy at line rate.

There is an open problem in *pHeavy* that is worth exploring further. The data plane is hard to install machine learning models that are trained by large dataset (e.g., several days network traffic), due to the large size of decision trees. A solution is to split large traffic into several parts based on periods of time, and the controller (e.g., P4 runtime [6]) can dynamically configure and modify machine learning models.

References

[1] *Data Set for IMC 2010 Data Center Measurement*. Accessed: Jun. 20, 2021. [Online]. Available: http://pages.cs.wisc.edu/ tbenson/IMC10_Data.html
[2] *Dpkt 1.9.2 Documentation*. Accessed: Jun. 20, 2021. [Online]. Available: https://dpkt.readthedocs.io/en/latest/
[3] *Imblearn.Under_Sampling.RandomUnderSampler*. Accessed: Jun. 20, 2021. [Online]. Available: https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.under_sampling.RandomUnderSampler.html
[4] *Intel Tofino Series Programmable Ethernet Switch ASIC*. Accessed: Jun. 20, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html
[5] *p4lang/Behavioral-Model*. Accessed: Jun. 20, 2021. [Online]. Available: https://github.com/p4lang/behavioral-model
[6] *p4lang/PI: An Implementation Framework for a P4Runtime Server*. Accessed: Jun. 20, 2021. [Online]. Available: https://github.com/p4lang
[7] *Scikit-Learn Machine Learning in Python*. Accessed: Jun. 20, 2021. [Online]. Available: https://scikit-learn.org
[8] *Tcpreplay—PCAP Editing and Replaying Utilities*. Accessed: Jun. 20, 2021. [Online]. Available: https://tcpreplay.appneta.com/
[9] *UNIBS: Data Sharing*. Accessed: Jun. 20, 2021. [Online]. Available: http://netweb.ing.unibs.it/ntw/tools/traces/
[10] *Weka the Workbench for Machine Learning*. Accessed: Jun. 20, 2021. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/
[11] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, vol. 10, 2010, pp. 89–92.
[12] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1172–1185, Jun. 2020.
[13] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Optimal elephant flow detection," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2017, pp. 1–9.
[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol.*, 2011, pp. 1–12.
[15] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-network inference with random forests," 2019. [Online]. Available: arXiv:1909.05680.
[16] S.-C. Chao, K. C.-J. Lin, and M.-S. Chen, "Flow classification for software-defined data centers using stream mining," *IEEE Trans. Services Comput.*, vol. 12, no. 1, pp. 105–116, Feb. 2019.
[17] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 254–265.

[18] P. Domingos, "MetaCost: A general method for making classifiers cost-sensitive," in *Proc. ACM SIGKDD Conf.*, 1999, pp. 155–164.
[19] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the accuracy of the ground truth associated with Internet traffic traces," *Comput. Netw.*, vol. 55, no. 5, pp. 1158–1167, 2011.
[20] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 245–256, 2004.
[21] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 339–350.
[22] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational IP networks: Methodology and experience," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 265–279, Jun. 2001.
[23] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. C. Claffy, "GT: Picking up the truth from the ground for Internet traffic," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 5, pp. 12–18, 2009.
[24] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.
[25] N.-F. Huang, G.-Y. Jai, H.-C. Chao, Y.-J. Tzang, and H.-Y. Chang, "Application traffic classification at the early stage by characterizing application rounds," *Inf. Sci.*, vol. 232, pp. 130–142, May 2013.
[26] Y.-H. Huang, W.-Y. Shih, and J.-L. Huang, "A classification-based elephant flow detection method using application round on SDN environments," in *Proc. 19th Asia-Pac. Netw. Oper. Manag. Symp. (APNOMS)*, 2017, pp. 231–234.
[27] N. Japkowicz, "Assessment metrics for imbalanced learning," in *Imbalanced Learning: Foundations, Algorithms, and Applications*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 187–206.
[28] L. A. Jeni, J. F. Cohn, and F. De La Torre, "Facing imbalanced data–recommendations for the use of performance metrics," in *Proc. Humaine Assoc. Conf. Affect. Comput. Intell. Interact.*, 2013, pp. 245–251.
[29] M. Kubat and S. Matwin, "Addressing the curse of imbalanced training sets: One-sided selection," in *Proc. ICML*, vol. 97, 1997, pp. 179–186.
[30] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, 2004, pp. 201–206.
[31] V. López, A. Fernández, and F. Herrera, "On the importance of the validation technique for classification with imbalanced datasets: Addressing covariate shift when data is skewed," *Inf. Sci.*, vol. 257, pp. 1–13, Feb. 2014.
[32] T. T. T. Nguyen and G. Armitage, "A survey of techniques for Internet traffic classification using machine learning," *IEEE Commun. Surveys Tuts.*, vol. 10, no. 4, pp. 56–76, 4th Quart., 2008.
[33] P. Poupart *et al.*, "Online flow size prediction for improved network routing," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, 2016, pp. 1–6.
[34] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang, "SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws," in *Proc. 43rd Allerton Conf. Commun. Control Comput.*, 2005, pp. 1–10.
[35] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Elsevier, 2014.
[36] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, 2017, pp. 164–176.
[37] B. Wang and J. Su, "A survey of elephant flow detection in SDN," in *Proc. 6th Int. Symp. Digit. Forensic Security (ISDFS)*, 2018, pp. 1–6.
[38] P. Xiao, W. Qu, H. Qi, Y. Xu, and Z. Li, "An efficient elephant flow detection with cost-sensitive in SDN," in *Proc. 1st Int. Conf. Ind. Netw. Intell. Syst. (INISCom)*, 2015, pp. 24–28.
[39] X. Zhang, L. Cui, K. Wei, F. P. Tso, Y. Ji, and W. Jia, "A survey on stateful data plane in software defined networks," *Comput. Netw.*, vol. 184, Jan. 2021, Art. no. 107597.

Xiaoquan Zhang was born in 1993. He received the B.E. degree in computer science and technology from the Chengdu University of Technology, China, in 2016. He is currently pursuing the postgraduate degree with Jinan University. His current research interests includes stateful data plane/programmable data plane, software-defined networking, and machine learning in computer network.

Lin Cui (Member, IEEE) received the Ph.D. degree from the City University of Hong Kong in 2013. He is currently a Professor with the Department of Computer Science, Jinan University, Guangzhou, China. He has broad interests in networking systems, with focuses on cloud data center networking, software defined networking, NFV, programmable networking, and distributed systems.

Fung Po Tso (Senior Member, IEEE) received the B.Eng., M.Phil., and Ph.D. degrees from the City University of Hong Kong in 2006, 2007, and 2011 respectively. He is currently a Senior Lecturer with the Department of Computer Science, Loughborough University. His research interests include: network policy management, network measurement and optimisation, service chaining, data center networking, software defined networking, and edge computing.

Weijia Jia (Fellow, IEEE) is currently a Chair Professor and the Director of BNU-UIC Institute of Artificial Intelligence and future Networks, and the VP for Research of BNU-HKBU United International College. He has over 500 publications in prestigious international journals/conferences and research books and book chapters. His research interests include smart city, IoT, knowledge graph constructions, multicast and anycast QoS routing protocols, wireless sensor networks, and distributed systems.

An Accurate & Efficient Approach for Traffic Classification Inside Programmable Data Plane

Muhammad Saqib*, Zakaria Ait Hmitti*, Halima Elbiaze*, Roch H. Glitho†

* Université du Québec à Montréal, Montreal, Canada
† Concordia University, Montreal, Canada
{saqib.muhammad,ait_hmitti.zakaria}@courrier.uqam.ca, elbiaze.halima@uqam.ca, glitho@ece.concordia.ca

GLOBECOM 2022 - 2022 IEEE Global Communications Conference | 978-1-6654-3540-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/GLOBECOM48099.2022.10000863

***Abstract*—In-network traffic classification is a class of in-network computing that brings significant benefits to the network, i.e., the first line of defence, classification at line rate and fast reaction time. However, it is still challenging to accurately and efficiently classify Internet traffic at an early stage due to a clear trade-off between flow identification time and classification accuracy - both are competing objectives. To this end, we introduce a framework that focuses on deploying an accurate network traffic classifier inside a programmable data plane that can classify the traffic at maximal speed while considering the underlying constraints of the device. Notably, we move from statistical feature-based traffic analysis and argue that traffic flow can be classified using a single feature called sequential packet size information as input. We evaluate our approach by identifying different types of IoT traffic inside a programmable data plane. Our findings demonstrate that accurate and early-stage network traffic classification is achievable with minor use of networking device resources.**

***Index Terms*—machine learning, programmable data plane, P4, in-network computing, in-network classification**

I. Introduction

Generally, in-network classification is a class of In-Network Computing (INC) [1] and inspired by the reconfigurability of the match-action paradigm [2]. With the rise of INC [3], [4], the interest is rapidly growing to run Machine Learning (ML) algorithms inside Programmable Data Plane (PDP) [5]–[7]. Running ML models inside the networking device significantly impacts the network. First, switches offer very high performance. The latency through a switch is in the order of hundreds of nanoseconds per packet [5]. Second, the performance of distributed ML is bounded by the time required to get data to and from nodes. So, if a switch can classify the traffic at the same rate that it carries packets to nodes in a distributed system, then it will equal or outperform any single node.

Last but not least, the networking device can serve as the first line of defence by terminating unnecessary data close to the edge. As a result, it can help save energy, reduce traffic load on networking infrastructure, and improve user experience by lowering communication latency. Latency-sensitive applications will significantly benefit. Packet and flow are the two core objects to extract from headers and payloads in the network traffic classification process. The work in [6] investigated the trade-off between a per-packet or a per-flow based classification model. Per-packet model proved to be more efficient but less accurate, while the per-flow one is contrariwise. Since accuracy and efficiency are two competing objectives, the value of both cannot be overstated. The information needs to be aggregated from several packets in the per-flow case. Having these richer features lead to better training of the model. However, keeping up-to-date flow states in the memory is a resources intensive solution that needs extra memory and demand for complex operations to derive useful information from the aggregated data. A per-packet model is potentially more efficient that can classify the traffic at a line rate without updating any features in the memory. However, it does not offer aggregated measurements and the possibility of learning from temporal correlation to the model. As a result, due to a clear trade-off between detection time and classification accuracy, it is still challenging to perform accurate and early-stage network traffic classification.

In addition, there are certain limitations at PDP, such as the lack of support for complex operations and a limited amount of memory (tens of megabytes) to store many features for the incoming flows. Hence, setting up an upper bound on the performance of the traffic classifier inside PDP. Therefore, the design of a classification model that fits the constraints of PDP (e.g., no floating points, no loops, and limited memory) is challenging. Consequently, it is necessary to use the minimum number of features to reduce lookup and update overhead and identify the flow earliest stage while respecting the underlying constraints at the data plane.

This work aims to propose an accurate and efficient in-network traffic classification approach subject to the data plane constraints. Instead of using several statistical features of the flow, which are memory intensive and need complex operations, we only take a single feature as an input. Our evaluation results show that the proposed solution can identify the traffic type for several source applications at an early stage of the flow creation.

The remaining of this work is organized as follows. Section II presents the background information and our work position with the literature. Section III represents the proposed solution. The system validation is shown in Section IV. Finally, Section V represents the concluding remarks and future directions.

This work was fully supported by CHIST-ERA program under the "Smart Distribution of Computing in Dynamic Networks (SDCDN)" 2018 call.

II. Related Work

This section aligns our proposed solution with state of the art. In recent years, there has been a rising interest in research combining ML and networking. For instance, recent works such as [8] investigated the problem of ML-based traffic classification. However, few considered the data plane programmability facet. For instance, papers [5], [6] use statistical properties analysis of the flow for traffic classification but do not consider the limitations at PDP. Therefore, we discuss the commonly used flow classification techniques and highlight their deficiencies. Finally, we highlight the importance of an accurate and efficient network traffic classification approach in next-generation programmable networks.

A. Flow classification techniques

Statistical-based flow analysis is a widely used technique for distinguishing network traffic by identifying differences in statistical properties of the flows [8]. Several packets must be tracked to obtain more detailed information about the flow. Sampling approaches were used to select a few packets for each flow and send them to the control plane, which hosts the classifier [9]. The choice of sampling rate, on the other hand, is crucial because it is highly dependent on the application requirements. A low sampling rate may result in a high rate of miss-classification, whereas a high sampling rate may overwhelm the controller with additional traffic overhead. In addition, the main flaw of this approach, which does not incorporate the learning process, is the static construction of such model [8]. In a dynamic networking environment, this flaw has a significant impact on the model's performance. Moreover, feature such as inter-arrival time is under time-domain measurement. Instability is the main problem with a time-domain measurement that it is always prone to performance degradation in dynamic network conditions. As a result, statistical-based traffic classification approaches are limited in their ability to handle the dynamics in next-generation high-precision networks due to these robustness flaws.

B. P4-switch as a classification machine

Naturally, the switch acts as a classification machine. Upon receiving an object (a packet), the switch first extracts the relevant features from its headers, such as IP, port, protocol type, packet size, etc. The parser extracts these fields where each field is itself a feature. The switch keeps these extracted features inside a Packet Header Vector (PHV) and then applies the pipeline process to the vector. Based on such motivation, the work in [5] demonstrates the mapping of trained ML algorithms to reconfigure able match-action tables (RMTs) [10]. Generally, the training module generates the resulting outputs in a decision tree where the control plane API called P4Runtime embeds the outputs into the switch's RMTs. The authors validate their work by classifying IoT traffic based on some statistical properties of the flows; however, they did not examine accuracy or efficiency. Another work investigates a clear trade-off between traffic identification time and classification accuracy inside PDP [6]. Since packet and flow are the two core objects of classification decisions, it is hard to decide when picking one another. The per-flow based classification process is costly in terms of classification latency and resource consumption. Also, the demand for complex operations limits its applicability to PDP. On the other hand, there is no need to keep track of the packets in the per-packet case. Therefore, the resources overhead are not present anymore. However, it is hard to accurately identify the traffic class based on very little independent feature information provided by individual packets. Consequently, bringing accuracy efficiency to the network traffic classification process is still challenging and highly desirable in next-generation high-precision networks.

III. System Design

This section presents the necessary steps to deploy an ML model into a programmable networking device by using P4 language. Fig. 1 shows the high-level in-network classification architecture. The control plane characterises the source traffic and maps the resulted output into the data plane for online inference[1].

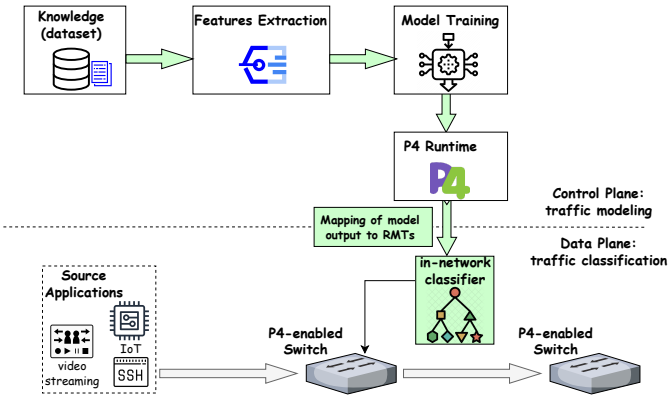


Fig. 1: In-network traffic classification architecture

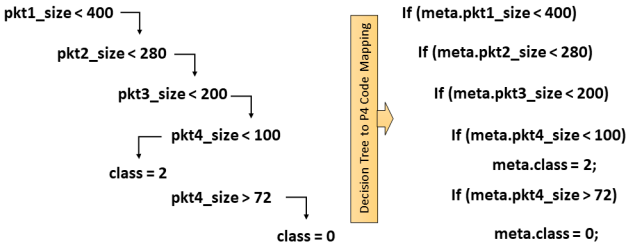


Fig. 2: Mapping of decision tree to P4 code

A. Offline data training

In this phase, the control counterpart trains ML models on a given dataset and translates them into target switches for traffic identification at runtime. This section focuses on ML model training while considering the requirements such as flow identification time, classification accuracy and limitations of P4.

Flow and metrics: A flow f_i is a sequence of packets p_j having the same five tuples (IP addresses, ports, protocol

[1]Our source code is publicly available at https://github.com/em-saqib/inc

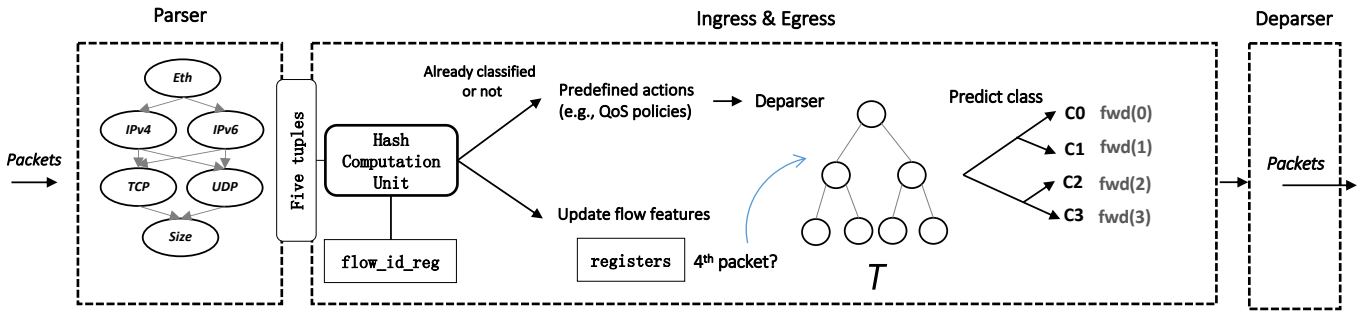


Fig. 3: Packet processing pipeline in the data plane

type). The first j packets of the flow f_i are denoted $f_i(1:j)$. The source dataset contains various traffic flows having different Quality of Service (QoS) requirements. Flows with similar QoS requirements belong to the same class, and the flow identification process for different QoS groups is known as multi-class classification. Two metrics are used to assess the flow classification accuracy: true positive rate (TPR) and true negative rate (TNR) [11].

Traffic features: TABLE II lists the features used to train ML models. The packet size is used for a single feature and the rest for multi-features model.

1) Feature extraction

The two core objects in network traffic classification are $flow_i$ and p_j, which are used to extract information from traffic headers and payloads. The flows are mainly identified based on the statistical properties of traffic. However, keeping up-to-date flows' states in the switch is a resources intensive solution that needs extra memory and demand for complex operations to derive useful information from the aggregated data. In addition, the features derived from time-related metrics such as inter-arrival time may not be consistent and stable enough to serve the classifier in dynamic network conditions [12]. Also, network conditions such as bursty heavy loads and traffic congestion may affect the time-related metrics. To solve this problem, we used the most stable feature, namely packet size information, as input for the classifier [13].

2) ML model training

The original dataset S_i consists of packets of subflows $(f_i(1:j))$ and is further split into training S_i^T and testing S_i^P samples. After preparing the dataset, the next step is to characterize the traffic by applying the ML algorithm. There are many supervised learning approaches in the literature, but not all are appropriate for our work. In other words, since we aim to embed the ML model's output into the data plane, the necessary operations in the targeted model must be readily available in P4. Therefore, we decided to use a decision tree algorithm to cope with the P4 limitations. Given the P4 language's current primitives, a decision tree classifier is more suitable for such a task. Only comparison operation is required to classify an element x, and it can be easily expressed in P4 using *if-else* statements (see Fig. 2). T_i is defined as a decision tree that predicts the flow class after receiving the j^{th} packet for a flow f_i.

3) ML to P4 converter

A decision tree algorithm makes decisions based on the values of input parameters (i.e., features) and can be represented with a tree structure. When using a decision tree to classify an element x, one must traverse the tree from root to leaf, respecting the conditions in each node until a leaf node is reached. This procedure can be easily implemented in general-purpose computer languages using recursion or repeating loops. In the P4 language, however, neither of these alternatives is available. As a result, hard-coding the tests and labels within the tree-nodes into *if* and *else* statements is an option. To that end, the ML to P4 component translates the model's *if-else* conditions into a P4 code that describes the generic behaviour for a given application.

B. Online inference

A general process of online inference is shown in Fig. 3. The switch maintains a few registers to record $flow\_id$, *packet_sizes*, *packet_counter* and other statistics such *min/max/avg packet size* and *total number of bytes* for the first few packets of each flow. The diagram depicts how each packet traversing the device is handled. For the incoming packets, the parser module extracts the relevant features (*i.e., five tuples and packet size*) from the header and keeps these features' values in the pipeline's metadata. The next step is to calculate the hash value for each flow based on the *five tuples* field from the packet's header. The *flow_id* register keeps track of all classified flows to treat the belonging packets accordingly. For the incoming packets belonging to identified classes, the switch applies corresponding actions. In other words, the classified flows' packets will not necessarily go through the decision tree process and will be processed at a line rate. In the event where the flow is not classified, the switch verifies *packet_counter* for the corresponding flow. Until the *packet_counter* reaches the *threshold*, the parser extracts *packet_size* and stores it into a *size_vector*. Once the *packet_counter* meets the *threshold*, a classification occurs on the *size_vector* in following the *if-else* chain, encoded in P4 code. The flow's class will eventually be saved in the *meta.class* variable. The detailed steps are revealed by

Algorithm 1: Online inference

```
Input: TCP and UDP packets, thr: max # of packets
Output: classes_vector
classes_vector = []; actions_vector = []; flow_id=[];
 size_vector=[];
min_size= ∞ ; max_size=0; avg_size=0; total_bytes=0;
Function InferClass(packets):
    while packets do
        flow_id = hash(five_tuple);
        if isClassified(flow_id) then
            ApplyAction(flow_id);
        else
            if pkt_counter <thr then
                single_feature(flow_id, pkt_size, pkt_
                counter, size_vector) ;   // Algorithm 2
                multi_features(flow_id, pkt_size, min_size,
                max_size, avg_size, total_bytes) ;
                 // Algorithm 3
                pkt_counter ++
            if pkt_counter == thr then
                classes_vector[flow_id] =
                Apply_SF_Model(flow_id, size_vector);
                 classes_vector[flow_id] =
                Apply_MF_Model(flow_id, features_vector);
End Function
Function isClassified(flow_id):
    if classes_vector[flow_id]! = 0 then
        return True;
End Function
Function ApplyAction(flow_id):
    egress_port = actions_vector[flow_id];
End Function
```

algorithms (1-3).

Algorithm 2: Single feature

```
Input: flow_id, pkt_size, packet_counter, size_vector
Output: size_vector[flow_id]
size_vector[flow_id * 4 + packet_counter] = pkt_size;
Return size_vector[];
```

Algorithm 3: Multi features

```
Input: flow_id, pkt_size, min_size, max_size, avg_size,
       total_bytes
Output: min_size, max_size, avg_size, total_bytes
if pkt_size<min_size then
    min_size = pkt_size;
if pkt_size > max_size then
    max_size = pkt_size;
total_bytes = total_bytes + pkt_size;
if packet_counter == thr then
    avg_size = Extern_Division(total_bytes, thr);
Return min_size, max_size, avg_size, total_bytes;
Function Extern_Division(total_bytes, thr):
    division_result = total_bytes/thr;
Return division_result;
End Function
```

IV. Performance Evaluation

This section evaluates our proposed in-network traffic classification solution by considering a use case of IoT devices generating data traffic belonging to various QoS groups. We demonstrate the efficiency of our solution explicitly in accurately identifying the class of source devices at an earliest stage inside PDP with minimal usage of device resources.

We use packet capture (PCAP) traces of IoT devices released by [14] as our dataset. From the available dataset instances, we selected the PCAP files for nine days (from 22 to 30 Sep 2016), containing flows related to five applications comprising different IoT devices. Since we aim to identify the source devices based on a single feature (i.e., packet size), we only select the packet size feature for our ML model training which can be directly extracted from the header.

TABLE I: Dataset summary

Device	Class	# of flow	# of packets
Amazon Echo (AE)	Smart assistants	16788	270840
Security Camera (SC)	Cameras	1601	144187
Motion Detector (MS)	Smart home devices	6411	233329
Photo-Frame (PF)	Appliances	5439	23561
Weather Station (WS)	Sensors	979	11623

TABLE II: Selected features

Feature	Type	Short Description
SrcPort	Stateless	Source port
DstPort	Stateless	Destination port
Pkt_size	Stateful	Size of the packet
Min_pkt_size	Stateful	Size of the smallest packet
Max_pkt_size	Stateful	Size of the largest packet
Avg_pkt_size	Stateful	Average packet size of flow
Total_bytes	Stateful	Cumulative sum of IPv4 packet size

1) Dataset

We divide the monitored devices to five classes: static smart-home devices (e.g., motion detector), sensors (e.g., weather station), audio (e.g., smart assistants), video (e.g., security camera), and appliance (e.g., photo frame). We select classes that can be assigned to various QoS groups: from high bandwidth (video) and low latency (sensor) to best effort (others classes). The devices belonging to the same class sharing the same traffic characteristics. Therefore, we select only one device from each class for validation in the data plane. TABLE I shows a summary of the dataset for these selected devices.

2) Experimental setup

The experimental procedure starts from training the ML model on the given IoT dataset to embedding the trained model' outputs to the data plane for online inference. We also observe the CPU and memory overhead added by the ML model to the P4 switch during online inference.

In order to compare the performance of our single feature-based traffic classification approach with a statistical method, we train two different ML models based on the features described in TABLE II. The statistical properties allow the capture of flow dynamics (e.g., duration and cumulative packet size at a given moment). However, the downside is extra resources overhead and limited support of crucial operations (e.g., division and square root) in P4. Therefore, it is impossible to compute better descriptive measures directly (e.g., average, variance, and standard deviation) for time-varying features when working with flows. In a single feature case, we

can directly extract the packet size from the header and the only operation required to infer class is a value comparison.

The first step is to train ML models on selected samples from the dataset. We are using Python's scikit-learn [2] implementation of the decision tree classifier to build the models. The training set for both single and multi-feature models is the same. However, the feature extraction process differs in both cases. In addition, the input length (i.e., number of packets) and decision tree depth must be kept to a minimum to identify the flow class at the earliest possible stage. Therefore, we are observing the input length and depth of the tree to understand the impact on classification accuracy. The obtained results show that with a tree depth of four and an input length of four, the single-feature model provides good accuracy (99%) as shown in TABLE III. As a result, both parameters are set to four.

The next step of the experiment is to test the models in the data plane for online inference. We are applying both single and multi-feature models to all the packets in the test set. The data plane is implemented in P4, compiled with a target of behavioural model version 2 (BMv2) [15]. Moreover, another experiment step is to assess the CPU and memory overhead added by both models to the regular packet processing to accomplish the actions required for classification (feature extraction, updating feature values, and identifying the flow class).

TABLE III: Varying input length and tree depth

	Input Length				
Tree Depth	**1**	**2**	**3**	**4**	**5**
1	55%	55%	72%	72%	75%
2	60%	60%	76%	84%	84%
3	71%	71%	83%	91%	91%
4	73%	73%	73%	**99%**	99%
5	73%	75%	90%	99%	99%

TABLE IV: Classification results

	Single-feature model			**Multi-features model**		
Class	Precision	Recall	F1-Score	Precision	Recall	F1-Score
AE	0.99	1.00	0.99	1.00	1.00	1.00
MS	1.00	1.00	1.00	1.00	1.00	1.00
PF	0.99	0.99	0.99	1.00	0.96	0.98
SC	1.00	0.92	0.96	0.94	0.91	0.99
WS	0.98	1.00	0.99	0.98	1.00	0.99

3) Results

This subsection presents the obtained results, including the classification accuracy, the class identification time and the resources overhead added by ML models to the data plane.

Classification accuracy: TABLE IV summarizes the classification results of both single and multi-feature models using the following performance metrics: precision, recall and f1-score. It is clearly shown that both models performed similarly for class identification. The confusion matrix for online inference of both models is shown Fig. 4, with the accuracy of both models being similar.

[2]https://scikit-learn.org/

Class identification time: This performance criterion concerns packet residence time inside the switch's pipeline. It can be obtained by calculating the packet forwarding latency throughout the pipeline processing. Equation 1 is used to calculate the identification time T_{f_i} of a particular flow f_i. It is the sum of j^{th} packet processing time and the time spent in feature extraction and values updating process by unclassified packets $(1 : j-1) \in f_i$.

$$T_{f_i} = T_t^{j \in f_i} + T_t^{(1:j-1) \in f_i} \tag{1}$$

The average flow identification time for each class is shown in Fig. 5. In the multi-features case, the class identification time increases with the number of features. This can be explained by the necessary operations for calculating statistical properties and updating corresponding flow entries in the memory.

Classification cost: the last set of results concerns the cost of network traffic classification inside PDP in terms of CPU and memory consumption of the switch. Again, it is slightly higher in the multi-features case, indicating that an increase in the number of features directly impacts the device's memory and computational resources. (See TABLE V).

TABLE V: Classification cost

Model Type	**CPU**	**Memory**
Single feature	6.1%	0.39%
Multi-features	7.4%	0.49%

4) Discussion

The obtained results reveal that instead of classifying the flow based on statistical properties that are prone to network dynamics, affect the class identification time and adds-up extra resources overhead at the device; we can accurately and efficiently identify the network traffic only based on the sequential packet size information of the first few packets by each flow. Since we are using packet size information as an input, the applications having large-sized messages such as high-quality video streaming will cause fragmentation at the network layer, affecting the overall class identification time. However, as shown in TABLE VI, the payload size in emerging and latency-critical IoT applications is less than the maximum transmission unit (MTU). As a result, our proposed approach is potentially applicable to high precision networks where the latency-critical applications will benefit greatly. Meanwhile, taking a single feature as the input increases the system utilization in terms of CPU and memory consumption. In addition, despite having good accuracy, the one fit model becomes outdated due to the changing traffic pattern. At the same time, the network device memory needs to be carefully maintained. Therefore, it is vital to continuously monitor the network device, remove inactive match-action rules from the device memory, and use the telemetry data to train the model better to keep an updated model in the data plane. All these limitations and considerations are subject to our future work.

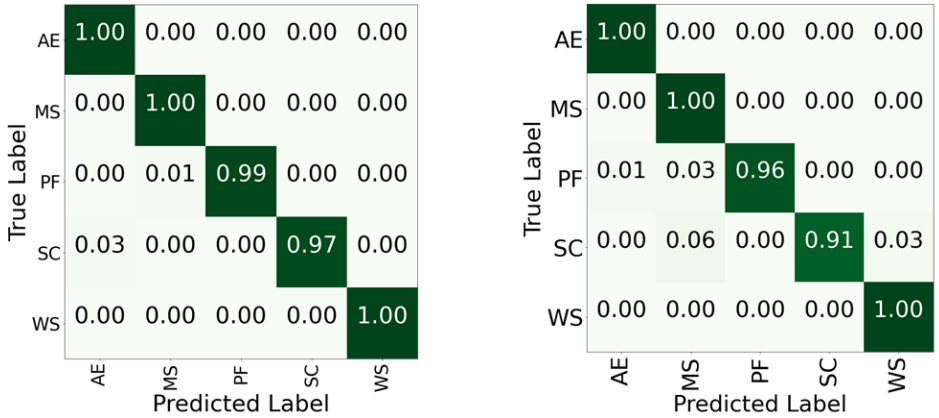


Fig. 4: Confusion matrices for single feature (left) and multi-feature (right) model

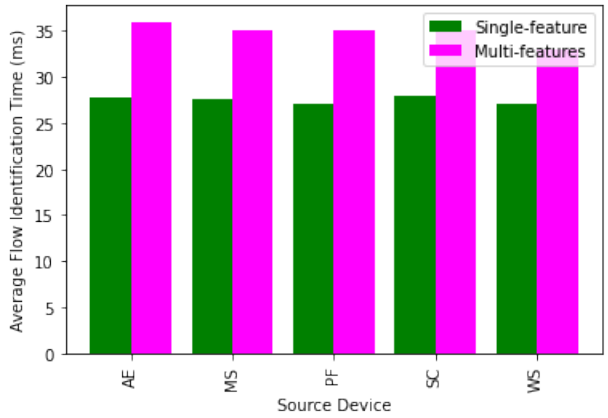


Fig. 5: Class identification time

TABLE VI: Payload size and latency requirements of IoT applications in next-generation networks [16], [17]

Use Case	E2E Latency (ms)	Data Size (bytes)
Factory Automation	0.5 to 50	10 to 30
Process Automation	50 to 100	40 to 100
Smart Grids / 2.0	3 to 20 / 1 to 10	80 to 1000
Intelligent Transportation	10 to 100	<500
Internet of Everything	ms to s	-

V. Conclusions

In this paper, we presented an accurate and efficient approach for network traffic classification inside PDP that can identify various types of IoT traffic at an early stage by considering only a single feature as an input. Our proposed solution mainly consists of two phases, (i) offline model training at the control plane and (ii) online inference in the data plane. The first phase uses a data training algorithm to identify the traffic source based on a single feature, while the second phase uses the resulted outputs as match-action rules inside PDP to identify the flow class at runtime. The simulation results show that accurate and early-stage network traffic classification is achievable inside PDP by considering only the first few packets' sizes information of each flow as an input. Our proposed solution achieves high precision early-stage network traffic classification, allowing efficient differentiated QoS provisioning.

References

[1] Y. e. a. Tokusashi, "The case for in-network computing on demand," in *EuroSys Conference*, 2019, pp. 1–16.
[2] P. e. a. Bosshart, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.
[3] I. Kunze, K. Wehrle, D. Trossen, and M.-J. Montpetit, "Use cases for in-network computing," *IETF, Internet-Draft*, 2021.
[4] Y. Tokusashi, H. Matsutani, and N. Zilberman, "Lake: the power of in-network computing," in *IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018, pp. 1–8.
[5] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *ACM workshop on hot topics in networks*, 2019, pp. 25–33.
[6] B. M. e. a. Xavier, "Programmable switches for in-networking classification," in *IEEE Conference on Computer Communications*, 2021, pp. 1–10.
[7] X. e. a. Zhang, "pheavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.
[8] F. e. a. Pacheco, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2018.
[9] S. Sadrhaghighi, M. Dolati, M. Ghaderi, and A. Khonsari, "Flowshark: Sampling for high flow visibility in sdns."
[10] P. e. a. Bosshart, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[11] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. IEEE, 2016, pp. 1–6.
[12] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson, "Offline/realtime traffic classification using semi-supervised learning," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1194–1213, 2007.
[13] W. e. a. Chen, "Sequential message characterization for early classification of encrypted internet traffic," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 4, pp. 3746–3760, 2021.
[14] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying iot devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, 2018.
[15] "Performance of bmv2," https://github.com/p4lang/behavioral-model/blob/main/docs/performance.mdwhat-impacts-performance, online; accessed 213 February 2022.
[16] C. e. a. De Alwis, "Survey on 6g frontiers: Trends, applications, requirements, technologies and future research," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 836–886, 2021.
[17] P. e. a. Schulz, "Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.

Binary Neural Network with P4 on Programmable Data Plane

Junming Luo
School of Electronics and Communication Engineering
Guangzhou University
Guangzhou, China, 510006
luojunming556@163.com

Waixi Liu
School of Electronics and Communication Engineering
Guangzhou University
Guangzhou, China, 510006
lwx@gzhu.edu.cn

Miaoquan Tan
School of Electronics and Communication Engineering
Guangzhou University
Guangzhou, China, 510006
tmqdyx24@163.com

Haosen Chen
School of Electronics and Communication Engineering
Guangzhou University
Guangzhou, China, 510006
csen_y@163.com

2022 18th International Conference on Mobility, Sensing and Networking (MSN) | 978-1-6654-6457-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/MSN57253.2022.00155

***Abstract*—Deploying machine learning (ML) on the programmable data plane (PDP) has some unique advantages, such as quickly responding to network dynamics. However, compared to demands of ML, PDP have limited operations, computing and memory resources. Thus, some works only deploy simple traditional ML approaches (e.g., decision tree, K-means) on PDP, but their performance is not satisfactory. In this article, we propose P4-BNN (Binary Neural Network based on P4), which uses P4 to completely executes binary neural network on PDP. P4-BNN addresses some challenges. First, in order to use shift and simple integer arithmetic operations to replace multiplication, P4-BNN proposes a tailor-made data structure. Second, we use an equivalent replacement programming method to support matrix operation required by ML. Third, we propose a normalization method in PDP which needn't floating-point operations. Fourth, by using register storing the model parameters, the weights of P4-BNN model can be updated without interrupting the P4 program running. Finally, as two use-cases, we deploy P4-BNN on a Netronome SmartNIC (Agilio CX 2x10GbE) to achieve flow classification and anomaly detection. Compared to the N3IC, decision tree and K-means, the accuracy of P4-BNN has 1.7%, 3.4% and 47.7% improvement respectively.**

Keywords—binary neural network; P4; Programmable Data Plane; anomaly detection; flow classification

I. INTRODUCTION

In recent years, we have seen an increasing interest in deploying machine learning (ML) to solve network problems [1]. The software-defined networking (SDN) with flexible programming has also promoted the deploying of ML. However, deploying ML on control plane of SDN has some shortcomings: the long latency of control loop, and additional overhead required to obtain global view. For example, in the case of flow classification, FlowSeer [2] generates an overhead of 288kps and a latency >1.98s; efficient sampling and classification approach (ESCA) [3] generates an overhead of 215kps and a latency of 1.98s.

To keep pace with an ever-changing set of requirements, network device has rapidly become programmable across the board: from switches to network interface cards (NIC) to middle boxes. These data planes are referred as programmable data plane (PDP). PDPs, such as programmable switch (e.g., Intel Tofino) and smartNIC (e.g., Netronome Agilio CX 2x10GbE [4], Mellanox MCX654106A-ECA) provide very high performance with processing latencies that can reach nanoseconds. PDPs have some computing and memory resources to support deploying ML. Furthermore, PDPs are the closest place to packets. Thus, deploying ML on PDPs has some unique advantages, such as, quickly responding to network dynamics, and avoiding the traffic overhead caused by sending packets to the control plane or outside analyzers.

However, in order to achieve forwarding packets at line rate, PDPs hold limited operations and programming model (e.g., missing loop), and can only give little memory and computation resource to support application-specific tasks offloaded. PDPs do not support complex operations required by ML, such as multiplication, polynomial or logarithms. Therefore, how to deploy ML on PDPs is a challenge.

While programmable devices have been proven to be useful for in-network computing, there are few successful cases of implementing ML on PDPs. Considering the computing and memory requirements of ML, most researches about deploying ML on PDP are simple non-neural network(NN) ML. For example, flow classification [5] and anomaly detection [6] by decision tree, support vector machine, clustering and random forest. But these simple non-NN ML cannot achieve satisfactory performance, such as low classification accuracy.

Ideally, the performance of NN with more layers is better than that of non-NN ML. However, compared to non-NN, NN require larger memory, more operations and programming model. At the cost of sacrificing little precision, binary neural networks (BNN) [7] require low memory, and its computation can be performed using lighter mathematical operations, such as bit shift. For programmable switch and smart-NIC, programming protocol-independent packet processors (P4) is current the most widespread abstraction and programming language. Specially, P4 program has good portability and can be ported across hardware platforms. Thus, this article proposes P4-BNN (Binary Neural Network based on P4). Our contributions are follows:

(1). We propose a framework that uses P4 to completely execute binary neural network on PDPs. Where we validate our framework by flow classification and anomaly detection use-case which are deployed on a Netronome smartNIC. For comparison, we also implement N3IC, decision tree by if-else chain and K-means with Manhattan distance on this smartNIC.

(2). We propose a tailor-made data structure suitable for deploying BNN on PDP, such that the input layer can use bit shifts and simple integer arithmetic operations to replace multiplication. Where the input of decimal features is converted to be binary. All 1-bit weights in a neuron are spliced into a

978-1-6654-6457-4/22/$31.00 ©2022 IEEE
DOI 10.1109/MSN57253.2022.00155

decimal value, which is stored by the registers to reduce the times of reading/writing registers.

(3). We propose a normalization method in PDP which needn't floating-point operations. Its key idea is that only caring about positive or negative of normalized results rather than their specific value.

(4). We propose a method of updating parameters of BNN model in runtime, where registers are used to store the weight parameters of the BNN model. Since the registers can be conveniently and quickly read and written from the control plane, the model can be updated in runtime.

II. Related Work

Non-NN on PDPs. IISY [8] introduced a network packet classification system on PDP. They explore packets classification using in-network supervised and unsupervised ML algorithms (decision trees, K-means, SVM, and Naïve Bayes) implemented in P4. [9] proposed to deploy a decision tree into the programmable data plane by matching action table for flow classification, but both of them use a lot of memory to store the table. [10] proposed an ML component in the control plane to convert the decision tree model to P4 language by if-else chain and deployed it into Netronome smartNIC to actually measure resource consumption. [11] proposed to deploy the random forest algorithm by matching action table in the programmable data plane to realize the task of attack detection. The stateful and stateless features with more complex calculation are used, but the performance results depend on the number of registers or other storage elements. DeepMatch [12] allow stateless intra-packet and stateful inter-packet (i.e., flow-based) deep packet inspection (DPI) on the NPs-based smartNIC, employing regular expression matching. Instead of only packet header inspection shown in other works, DeepMatch delivers line-rate regular expression matching on packet payloads.

NN on PDPs. N2Net [13] and BaNaNa [14] have made significant contributions to offloading deep learning (DL) processing in the programmable data plane. N2Net designed for embedded applications that run on resource constrained devices and thus require simple arithmetic operations. However, it is not deployed and tested in the real network. BaNaNa proposes to spilt the input layers of NN on a CPU. The remaining layers go through a quantization process that converts the original NN model into a format that can be run on programmable network devices. However, this method adds communication latency between CPU and PDP. N3IC [15] use Micro-C language to implement the neural networks on Netronome SmartNIC with simple bit operations, but it has not yet provided normalization layer for neural networks.

We completely converted the BNN into the P4 language, and deployed all neural network layers onto PDP, and provide a more detailed process of deploying BNN on PDP.

III. Background and Motivation

A. SmartNIC

The NIC communicates with the host's CPU and GPU through PCIe. This process involves four times PCIe transfers, which often takes a lot of time to transmit data in delay-sensitive network tasks. Some lightweight tasks can be offloaded to the smartNIC, which not only reduces the data processing latency, but also releases the CPU resources of the host.

B. Binary Neural Networks

(1). Equation (1) is the standard for the binarization of weight parameters. A weight value (w) occupies only one bit. Compared to the floating-point weight matrix, the memory consumption of the network model can be reduced 32 times than before.

$$\text{sign}(w) = \begin{cases} +1, & w \geq 0 \\ -1, & \text{others} \end{cases} \quad (1)$$

(2). When the weight value and the activation function value are binarized at the same time, the multiplication and addition operation of 32 floating-point numbers can be solved by the simple operation of bitwise (XNOR), and population count (POPCOUNT).

We conducted validation experiments to evaluate the compressed model size of BNN and the computational complexity of the inference process. For MNIST dataset to use LeNet-5 model, before binarization, the model size is 173KB and the accuracy reaches 98.6% during the inference process, and the calculation amount is 282KMACs (MAC represents a Multiply-accumulate operations). After binarization, the model size is reduced to 7KB and the accuracy can also reaches 97.0%. More important, the multiplication operation in the inference process can be replaced by simple arithmetic operations of bit.

IV. P4-BNN System Design

In this section, we describe the challenges of deploying DL on PDP, and how do P4-BNN to address them.

A. Challenges of Neural Networks on PDP

Challenge 1. Small model is desired. A typical smartNIC has ~10 MB memory, which is mainly used to store packet information, and host forwarding and policy tables. Thus, the memory left for NNs is very small. However, NNs often have lots of parameters, which require a large of memory.

Challenge 2. Simple operations are desired. The computing capabilities of network devices is limited. PDPs often only perform bitwise logic, shift and simple integer operations. However, NNs require a lot of complex operations (e.g., matrix operations, multiplication, division, floating-point, and loop).

B. P4-BNN overview

The P4-BNN architecture is shown in Fig.1, which converts the NN inference into P4 and is deployed on a smartNIC to achieve flow classification and anomaly detection. Where all operations for flows are done on the data plane instead of the control plane. This can avoid latency caused by sending flows information to the controller.

Based on P4-BNN, we design two types of classification models. One is the packet-level model which employs features from individual packet, and another is flow-level which employs features from flows. Specially, one advantage of flow-level model is the ability of capturing the continue network dynamics, such as cumulative flow duration and flow length at a given moment, which cannot be accomplished only relying on single packets. They have richer features, so the flow-level model

performance will be better than that of packet-level. However, the flow-level model requires retaining each flow information to store their respective feature values. The packet-level model need not store anything, so there are no overhead associated with flow information, such as memory consumption. However, packet-level model requires the ML to process each packet, and this will may lead to long processing latency. These infers will be confirmed by the latter experiment results.

Control Plane. It is responsible for collecting flow information from the data plane to build training dataset and training the BNN model. This dataset is processed into binary encoding (Section IV-C) for training. After the model training, the control plane send new model parameters to the smartNIC's registers to update the weight of the model.

Data Plane. It receives commands and parameters from the control plane, such as accepting new model parameters. When the packet enters the smartNIC, it extracts packet information from the packet header and then performs BNN inference, normal packet forwarding, and updates flow information.

When a packet enters the smartNIC, for flow-level model, the processing includes the following four steps. For packet-level model, only the third and fourth steps need to be performed, but BNN operations are done for each packet, as shown in Fig.1.

(1). **Hashing**. We use hash to mark each packet with the flow ID which it belongs to, and then look up the *result register* according to the flow ID. Where the inputs of hash are some header fields (i.e., IP.src, IP.dst, Port.src, Port.dst, IP.protocol). If the classification result exists in the *result register* (The classification results are obtained according to the first *N* packets of the flow), it will be written to the ToS field of IP header for forwarding. Otherwise, next step is the *feature gathering*.

(2). **Feature gathering**. It extracts the relevant features of the packet and stores them into *feature register*. *N* is a threshold value that indicates the number of packets needed to be collected. If the counter reaches the *N*, the *BNN executor* is triggered to perform the BNN inference. Otherwise, the packet will be normally forwarded without classification result.

(3). **BNN executor**. It executes BNN inference to classify flow. The classification process is as follows. First, the *feature register* is read to get flow features, and then these features are inputted to the BNN model for inference. Finally, the inference result is assigned to the flow and used for updating *result register*. The classification result is written into ToS field of IP header.

(4). **Forwarding**. According to the classification result, selecting different ports to forward the packets.

C. Design Details of P4-BNN

(1). Achieving P4-BNN Inference on PDP

The P4-BNN inference process is shown in Algorithm 1. The basic operations are XNOR, POPCOUNT, NORMALIZATION and SIGN. The computation between neurons use XNOR operation which is a simple shift. SIGN is a comparison operation. NORMALIZATION is an operation without floating-point, as shown in line 4 to line 13. For POPCOUNT, we count the number of 1 in bit string. Specially, at line 3, the $W_k^b \oplus a_k$ represents XNOR between the binary weight of b^{th} neuron at k^{th} layer and input of k^{th} sample to obtain a bit string. Then performing POPCOUNT operation to obtain the number of 1 in the bit string (*i.e.*, *c*). Next, performing NORMALIZATION operation to make the linear activation value in the same distribution (*i.e.*, *d*). Finally performing SIGN operation to obtain the final result (*i.e.*, SIGN(*d*)) which is a value of 1 bit (0 or 1). Because there are multiple neurons, this operation is repeated for many times, in other words, there are multiple *d*. We splice these SIGN(*d*) into a bit string as the input of next layer, as shown in line 14 of Algorithm 1. More details are shown in *calculation between neurons* of Fig. 2.

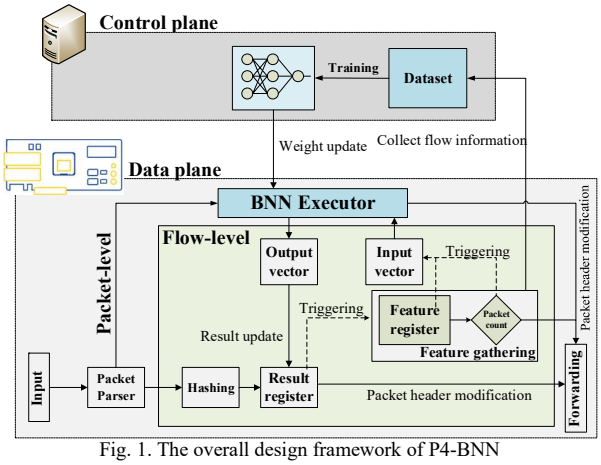


Fig. 1. The overall design framework of P4-BNN

Algorithm 1 P4-BNN Inference.

Input:
a_k : binary input of k^{th} layer
W_k^b : binary weights of b^{th} neuron at k^{th} layer
$\lceil \mu_k \rceil$: mini-batch mean of k^{th} layer
Output:
y : prediction results

1: for k = 1: K-1 do //K is the number of layers
2: for b = 1 : B do // B is the number of neurons in each layer.
3: c $\leftarrow$ POPCOUNT($W_k^b \oplus a_k$) //c is the number of 1 in the bit string, $\oplus$ is XNOR operation.
4: if $c \geqslant t_k$ //t_k is half the length of the bit string.
5: if *sign* ==0 // *sign* is the sign bit of $\lceil \mu_k \rceil$
6: $d = 2 * (c - t_k) - \lceil \mu_k \rceil$
7: else
8: $\mathrm{SIGN}(d) = 0$
9: else
10: if *sign* ==0
11: $\mathrm{SIGN}(d) = 0$
12: else
13: $d = |\lceil \mu_k \rceil| - 2 * (t_k - c)$
14: $a_{k+1} \leftarrow (a_{k+1} \ll 1) + \mathrm{SIGN}(d)$
15: end for
16: y $\leftarrow$ SIGN(POPCOUNT($W_K^1 \oplus a_K$))

(2).Tailor-made Data Structure and Matrix Equivalent Replacement

The inference of BNN can use XNOR and POPCOUNT instead of multiplication, but this method is only suitable for hidden layers. For the input layer, the input is usually in decimal form. For such inputs, XNOR and POPCOUNT operations cannot be used to speed up the inference process. In other words, the non-match between input layer and hidden layers maybe

slow down the speed of inference. Therefore, P4-BNNprocess input data by binary, i.e., converting the value of each feature from decimal to be binary. For example, the source ports and destination ports are a 16-bit integer in the PDP, as shown in *tailor-made data structure* of Fig. 2. This processing turns the one original input into multiple inputs. While this will increase the number of inputs, it can let the input layer to be accelerated by simple operations, which naturally match with the characteristics of PDP.

Each weight of P4-BNN model has only 1 bit with values 0 or 1 (P4-BNN uses 0 instead of -1). Thus, during inference, we use registers to store the weights of model. As shown in *Matrix Equivalent Replacement* of Fig. 2, to reduce the times of reading/writing registers, all 1-bit weights in a neuron are spliced into a decimal value stored by register. Each value of register represents the weight of a neuron. Besides, in normal programming, matrix operation can be realized by loop. However PDP does not support loop operation, we use multiple reading/writing registers to equivalently replace matrix operation between neurons. The times of reading /writing registers are same as the number of neurons.

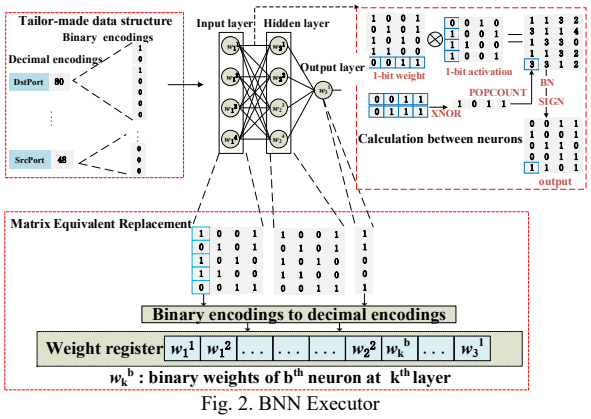


Fig. 2. BNN Executor

(3). Batch Normalization in PDP

In terms of accelerating and stabilizing the training of neural networks, it has been proved that the normalization is an effective technology . However, the floating-point operations required in the normalization make it difficult to be implemented in PDP. Batch normalization (BN) [16] use trainable parameters γ and β to retain the nonlinear features learned from activation. We observe that BN in P4-BNN can eliminate floating-point operations by set γ=1 and β=0. In the hidden layers, subtracting the mini-batch mean (μ) from the linear activation value is the main operations of normalization, and then the result of subtraction will be divided by the root of the mini-batch variance ($\sqrt{\sigma^2+\varepsilon}$). Therefore we only need to care about the positive or negative of normalized results rather than their specific values which will be binarized (0 or 1). Because $\sqrt{\sigma^2+\varepsilon}$ is always positive, the division operation can be ignored. We use ⌈μ⌉ to approximate μ, where the sign is the highest bit.

In summary, BN includes three steps, (i) checking whether the c is greater than the threshold (t); (ii) checking the sign bit of the ⌈μ⌉; (iii) linear activation value subtract the ⌈μ⌉ as the input of SIGN. As shown in line 4-line 13 of Algorithm 1.

(4). Updating BNN Model in Runtime

Because the traffic pattern vary greatly, the classification performance maybe worse if a unchanged BNN model is used to classify these dynamic traffic. Therefore, to improve the generalization of P4-BNN model, how to update model in runtime is an important issue. In the P4-BNN, the control plane can conveniently and quickly read and write the registers, so P4-BNN can real-time collect new packet features by reading the features registers and uses some registers to store the weights of BNN model. Thus, after training a new BNN model, the control plane can update the BNN model by modifying the weight stored in the register (i.e., writing the registers), when the P4 program is running.

V. Use Case

Flow classification. UNI1 [17] is a widely used dataset to evaluate flow classification. The flow types can be classified from the dimensions of flow length, flow duration, flow speed, etc. This article uses the flow length as the classification basis. For UNI1, we set the threshold of flow length to 10KB. So, when the flow length is greater than 10KB, it is an elephant flow, otherwise it is a mouse flow. According to this criterion, we sorted out 277130 flows, including 63298 elephant flows and 213832 mouse flows. After splitting the dataset, 80% of the flows is used for training and 20% is used for testing. The proportions of flows in each class were set to be the same as in the whole dataset.

For flow classification, we also uses the flow length and flow duration as the classification basis to achieve four classification. If the time interval between two packets in the same flow is greater than 60s, the following packets will form a new flow. According to this criterion, we observe that most of the flow length are less than 2KB and most of the flow duration are less than 1s. Table I shows the threshold used when building four-classifications.

Anomaly detection. The CIC-IDS2017 [18] contains benign and the most up-to-date common attacks, which resembles the true real-world data (PCAPs). This includes network traffic from Monday to Friday. We selected the PCAP files from Wednesday and divide it into attack and benign network flow. we sorted out 60000 data flows, including 30000 attack flows and 30000 benign flows. After splitting the dataset, 80% of the flows is used for training and 20% is used for testing.

Features analysis. How to select the suitable features is one key issue for flow classification and anomaly detection . We determine features by considering the cost of PDP extracting feature and scoring base on random forest. Where the higher score it gets, the better the relation between the feature and flow type [19]. Table II shows the features used when building packet-level model and flow-level model in P4-BNN. For flow-level four classification, we added flow duration as feature.

TABLE I. Threshold of four types of flows in UNI1

	Flow length	Flow Duration	Number of flows
Class 0	⩽ 2KB	⩽ 1s	133411
Class 1	⩽ 2KB	> 1s	42884
Class 2	> 2KB	⩽ 1s	62987
Class 3	> 2KB	> 1s	37848

TABLE II. PACKET-LEVEL AND FLOW-LEVEL FEATURES

Model	Feature	Short description
Packet-Level	DstPort	TCP/UDP destination port
	SrcPort	TCP /UDP source port
	Packet length	Length of the packet
	TTL	Time to live
Flow-Level	DstPort	TCP/UDP destination port
	SrcPort	TCP/UDP source port
	Packet length	Length of the packet
	Flow length	Length of the flow since the arrival of the first packet

VI. EXPERIMENTS

A. Experiment setup

Testbed. We benchmark P4-BNN in a simple two node topology. The Netronome smartNIC (Agilio CX 2x10GbE) deployed P4-BNN is installed in a Sugon I420 server with dual Intel Xeon 420 10-core 2.60 GHz processors and 16 GB DDR3 1600MHz RAM. It is connected via 10 GbE cables to a traffic generation server. The traffic generation server is also a Sugon I420. The traffic generation server uses *tcpreplay* command to send these packets to the virtual interface vf0-1.

Inside the smartNIC, each packet was handled according to three versions of P4 application. (i) Baseline, no ML algorithm is deployed; (ii) ML model of packet-level; (iii) ML model of flow-level. Each model structure of P4-BNN is a regular Multilayer Perceptron (MLP) with 3 fully connected layers (FC) of 8, 16, 1, where also with 2 normalization layers. More details are shown in Table III.

TABLE III. DETAILS OF P4-BNN MODEL STRUCTURE

Class	Input size (bits)	NN size (neurons)	Memory (KB)
Packet-Level	56	8,16,1	0.258
Flow-Level	128	8,16,1	0.338

Comparison. We realized an ablation about the true effect of batchnormalization in the scene (N3IC [15]), the decision tree (DT) with an if-else chain [10] and K-means measured by Manhattan distance on smartNIC are compared with P4-BNN. Of course, DT and K-means can also be realized with the match-action table, but it requires a lot of memory. Besides, full precision model based on artificial neural network (ANN) achieved on server CPU is regarded to a performance baseline, which model struct is same as P4-BNN.

B. Experiment results

Table IV contains accuracy, precision and recall on the flow classification and anomaly detection, where we observe the first 5 packets(i.e., N=5) in the flow. We observe that the ANN has the best performance, as expected. Our proposed P4-BNN method has only slightly lower accuracy than ANN. Because binarization loses the information carried by the weights, but the P4-BNN can compress the memory required for weights by 32 times compared with the ANN. P4-BNN performs better than N3IC, because the batch normalization layer can improve the accuracy of the neural network. We can also observe that whether packet-level or flow-level, P4-BNN shows higher performance than DT and K-means model. Take flow classification as an example, we can see that the accuracy of the flow-level is higher than the packet- level. Because the flow-level contains the information of multiple packets, it has richer features.

Furthermore, we also evaluate P4-BNN for four-classification task. Table V shows that the accuracy of ANN, P4-BNN, N3IC and K-means all greatly decrease. The behind reason is follows, the four-classification task requires more feature/weight information than two-classification task, however the weight of P4-BNN lose too much information after binarization. When P4-BNN use a same model structure, fitting the data for four-classification task is obviously more difficult than two-classification task. To address this challenge from four-classification task, designing a more complexly model structure for P4-BNN maybe a possible method, and we will do it in the future.

TABLE V. RESULTS OF FLOW-LEVEL FOR FOUR CLASSIFICATION.

Method	Accuracy	Precision	Recall	F1
ANN	77.3%	72.3%	70.4%	0.713
P4-BNN	55.4%	44.1%	46.9%	0.455
N3IC	48.8%	37.1%	26.2%	0.307
K-means	48.2%	34.7%	25.2%	0.292

C. Overhead

Packet processing latency. In order to evaluate the packet processing latency added by deploying ML, we measure the time difference between packet enters the physical port of smartNIC and the packet reaches virtual interface vf0-1. Fig. 3 shows the distributions of the time that a packet resides inside the smartNIC according to flow-level and packet-level for flow classification. We can see that the packet processing latency has increased after ML algorithm are deployed on the smartNIC.

The processing latency of the P4- BNN packet-level model is the largest. The main reason is that although P4-BNN packet-level classification does not require updating flow information and storing any data, but it will require operations for each packet (XNOR, POPCOUNT, SIGN and BN). The overhead of P4-BNN flow-level model is mainly caused by updating flow information. After collecting enough features for P4-BNN model, the remaining packets do not need to execute the P4-BNN inference process, but only need to find the result register to get the classification result. It can be seen that the overhead of P4-BNN inference process is higher than that of updating flow information.

For flow-level, the overhead of DT, K-means and P4-BNN models are almost the same, because their overhead is mainly caused by updating traffic information. For packet-level, we observed that the overhead of P4-BNN is much higher than DT and K-means, because P4-BNN requires more complex operations rather than a simple if-else chain. K-means needs to calculate the Manhattan distance, so the processing delay will be larger than DT, but the calculation of the distance is not as complex as the inference process of P4-BNN, so the processing delay will be shorter than P4-BNN.

Flow classification scheme based on P4-BNN can help flow scheduling on switch. Compared with making decisions in the control plane [2, 3], the increased processing latency of P4-BNN can be ignored.

Memory of code. For the Netronome smartNIC Agilio CX 2x10GbE, the entire application must be described in 8K instructions (or 16K in shared-code mode). Table VI shows the amount rate of code instructions for flow classification. Where Local Memory (LM) and cluster local scratch (CLS) represent

TABLE IV. SUMMARY OF RESULTS

	Flow classification								Anomaly detection							
	Flow-Level				Packet-Level				Flow-Level				Packet-Level			
	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
ANN	92.9%	95.3%	95.6%	0.954	87.8%	75.7%	65.7%	0.734	96.6%	99.6%	93.4%	0.964	87.8%	92.1%	93.3%	0.927
P4-BNN	91.7%	93.6%	94.7%	0.942	85.8%	66.8%	57.6%	0.619	96.4%	99.5%	93.2%	0.962	86.8%	90.9%	93.5%	0.922
N3IC[15]	90.0%	88.6%	89.0%	0.888	84.6%	70.9%	45.7%	0.556	95.7%	99.6%	91.7%	0.955	85.5%	90.5%	92.2%	0.914
DT [10]	90.6%	94.1%	93.8%	0.939	81.6%	70.9%	35.2%	0.471	93.7%	99.9%	87.4%	0.932	83.4%	83.5%	99.8%	0.909
K-means	71.4%	96.8%	65.1%	0.777	65.5%	40.0%	95.8%	0.563	48.7%	47.5%	51.6%	0.495	49.5%	84.0%	48.7%	0.616

the memory space of the code in the smartNIC. We observe that DT and K-means has a low code instructions due to its simple implementation. P4-BNN needs more code instructions due to its simple implementation. P4-BNN needs more code instructions, due to it is more complex than DT and K-means. It needs to do some calculations like XNOR, POPCOUNT, SIGN and BN between neurons.

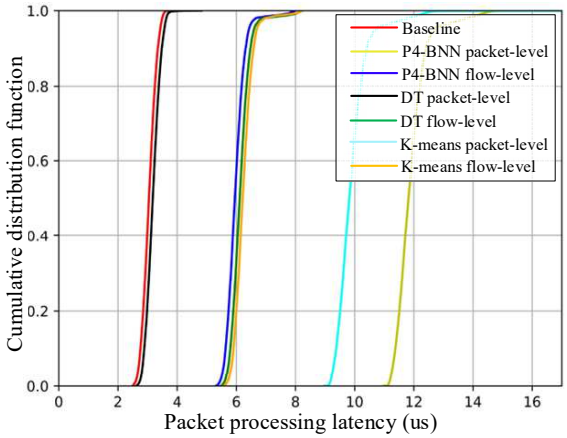


Fig. 3. Cumulative distribution function of packet processing latency at the smartNIC.

TABLE VI. THE AMOUNT RATE OF CODE INSTRUCTIONS

	Baseline	P4-BNN Packet-level	P4-BNN Flow-level	DT Packet-level	DT Flow-level	K-means Packet-level	K-means Flow-level
CLS	23%	28%	39%	23%	28%	27%	34%
LM	39%	49%	69%	39%	49%	49%	59%

VII. CONCLUSIONS AND FUTURE WORK

In this article, we propose a method using BNN for in-network classification, solving the problem of deploying neural networks in the programmable data plane, and deploy it into Netronome smartNIC to validate our ideas. The results show that in flow classification and anomaly detection, the accuracy reaches more than 91.7% and 96.4% respectively.

Furthermore, since single PDP is difficult to support full ML, a distributed deploy framework for ML maybe a promising method where distributing the neurons of an neural network (NN) into multiple switches. In this direction, in-network neural networks [20] has presented some interested use-cases, such as, network telemetry and anomaly/intrusion detection.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (61872102, 61972104, 62272113) the Guangdong Basic and Applied Basic Research Foundation (2021A1515012306, 2018A0303130045), Guangzhou Key Laboratory of Software-Defined Low Latency Network (202102100006), China.

REFERENCES

[1] R. Boutaba, M. A. Salahuddin, N. Limam, et al, “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities,” Journal of Internet Services and Applications, vol. 9, no. 1, p. 16, 2018.

[2] Chao S C, Lin K C J, Chen M S. "Flow classification for software-defined data centers using stream mining" [J]. IEEE Transactions on Services Computing, 2016, 12(1): 105-116.

[3] Tang F, Zhang H, Yang L T, et al. "Elephant flow detection and differentiated scheduling with efficient sampling and classification" [J]. IEEE Transactions on Cloud Computing, 2019:1-15.

[4] NETRONOME. (2019) Netronome Agilio SmartNIC. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[5] Eric Liang, Hang Zhu, et al. Neural Packet Classification. In Special Interest Group on Data Communication (SIGCOMM). ACM, 2019.

[6] Georgios Kathareios, Andreea Anghel, Akos Mate, et al. Catch It If You Can:Real-Time Network Anomaly Detection with Low False Alarm Rates. In International Conference on Machine Learning and Applications (ICMLA). IEEE,2017.

[7] Qin H, Gong R, Liu X, et al. Binary neural networks: A survey[J]. Pattern Recognition, 2020, 105: 107281.

[8] Zheng C, Xiong Z, Bui T T, et al. IIsy: Practical In-Network Classification[J]. arXiv preprint arXiv:2205.08243, 2022.

[9] Kamath R, Sivalingam K M. Machine Learning based Flow Classification in DCNs using P4 Switches[C]//2021 International Conference on Computer Communications and Networks (ICCCN). IEEE, 2021: 1-10.

[10] Xavier B M, Guimarães R S, Comarela G, et al. Programmable switches for in-networking classification[C]//IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE, 2021: 1-10.

[11] Lee J H, Singh K. SwitchTree: in-network computing and traffic analyses with Random Forests[J]. Neural Computing and Applications, 2020: 1-12.

[12] Hypolite, Joel, et al. "DeepMatch: practical deep packet inspection in the data plane using network processors." Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies. 2020.

[13] Siracusano G, Bifulco R. In-network neural networks[J]. arXiv preprint arXiv:1801.05731, 2018.

[14] Sanvito D, Siracusano G, Bifulco R. Can the network be the AI accelerator?[C]//Proceedings of the 2018 Morning Workshop on In-Network Computing. 2018: 20-25.

[15] G. Siracusano, S. Galea, D. Sanvito, et al. Rearchitecting Traffic Analysis with Neural Network Interface Cards, USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022.

[16] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C]//International conference on machine learning. PMLR, 2015: 448-456.

[17] Theophilus Benson, Aditya Akella, David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild[C]// Acm Sigcomm Conference on Internet Measurement. ACM, 2010.

[18] Sharafaldin I, Lashkari A H, Ghorbani A A. Toward generating a new intrusion detection dataset and intrusion traffic characterization[J]. ICISSp, 2018, 1: 108-116.

[19] W. Liu, J. Cai, Y. Wang, et al, “Fine-grained flow classification using deep learning for software defined data center networks,” Journal of Network and Computer Applications, 2020,168(10).

[20] Luizelli M C, Canofre R, Lorenzon A F, et al. In-Network Neural Networks: Challenges and Opportunities for Innovation[J]. IEEE Network, 2021, 35(6): 68-74.