AdaFlow: Efficient In-Network Traffic Classification using Programmable Switches

Sankalp Mittal
IIT Hyderabad, India

Harshith Kotha
IIT Hyderabad, India

M. Anand Krishna
IIT Hyderabad, India

Praveen Tammana
IIT Hyderabad, India

Abstract—**In-network ML-based traffic classification using programmable switches has enabled faster decisions and reduced security infrastructure's cost and management overheads. However, due to constraints on per-packet operations and limited stateful memory in the switch data plane, there is a fundamental tradeoff between classification accuracy and memory requirements. The existing work falls short of accurately classifying traffic with diverse flow characteristics while keeping the memory footprint low. In this paper, we propose `AdaFlow` system that aims to address this gap by incorporating traffic-specific heuristics while designing the in-network classifier. We evaluate `AdaFlow` prototype via simulations and also on a testbed with an Intel Barefoot Tofino switch. Compared to the state-of-the-art, `AdaFlow` improves accuracy while keeping the memory overheads similar to or lower than the existing systems.**

I. Introduction

Programmable switches have enabled line rate in-network traffic classification [1]–[6] by deploying decision tree based supervised ML models directly to the data plane. Compared to the approaches classifying traffic at the control plane [7], line rate traffic classification entirely in the data plane has enabled faster decisions [8] and reduces the cost of building and managing security infrastructure. However, due to limited switch memory (10s of MBs [9]) and compute restrictions there is a fundamental tradeoff between the line rate classifier accuracy and the switch data plane resources. Therefore, designing a memory-efficient classifier while maintaining accuracy for traffic with diverse flow characteristics is important and an active research area.

Some existing in-network classification systems perform inference using either packet-level (PL) features [1], [2], [10], [11] (*e.g.,* packet length, IP address) or flow-level (FL) features [3], [12]–[15] (*e.g.*, average packet size, minimum IAT, *etc.*). Using only PL features keeps data plane resource overheads low as there is no state to maintain across packets (on the other hand, FL features require maintaining state), but it has poor classification accuracy for tasks that require flow-level (FL) features [4]. Systems using only FL features would delay inference until the collected FL features are reliable enough to infer a traffic class. This not only makes them *insufficient* as they allow initial packets in a malicious flow to go through as benign but also inaccurate as flow state may get overwritten by new flows due to collisions before their features become reliable[1] and classified accurately [5].

[1]It has been found that *prematurely* classifying a flow based on FL features may often be worse than using PL features [5], [16].

On the other hand, there are systems [4], [5] that perform inference using both PL and FL features and handle flow-level collisions. But NetBeacon's [4] data plane module has high memory overhead as it uses multiple ML models. Though Jewel [5] is memory efficient by using a single unified ML model in the data plane, it uses only a single inference point[2], thus it leads to poor accuracy for traffic with diverse flow characteristics (a mix of short, long, benign, and malicious). See §VII for details.

In summary, the existing work falls short of an in-network classifier that *accurately* classifies traffic with diverse flow characteristics while keeping the *data plane memory footprint low.* Addressing this problem, in this paper, we propose a general, memory-efficient, and accurate in-network classifier called `AdaFlow` . `AdaFlow` achieves them by incorporating traffic-specific heuristics on flow characteristics while training ML models (§III) and while resolving flow-level collisions using a priority-based mechanism (§IV).

The main contributions of this paper are:

Multi-phase Aggregated ML model. Packets in a flow are located at different phases (*e.g.,* packet counter) so each phase has different FL feature values [4]. Upon collision with an existing flow, PL feature values can be used for inference. However, this requires separate ML models to be deployed for PL features and PL+FL features which have a high switch memory overhead. We address this problem by designing a single multi-phase aggregated ML model. The key idea is that the early packets of a flow use only PL features for inference (until a threshold n^{th} packet is reached) while updating FL features in parallel. After the packet count exceeds the threshold, *we switch to inference using both PL and FL features for every incoming packet* (instead of a few inference points) until the confidence score of inferred class $P(inferred\_class)$ exceeds a certain determination threshold $dt \in [0, 1]$. We realize this idea by carefully augmenting data while training ML models (more details in §III).

PrioritySketch algorithm. While resolving collisions in a memory-constrained data plane, giving equal priority to all flows would not give good classification accuracy across different traffic datasets because of the diversity in their flow characteristics. To address this problem, we design and

[2]A flow's packet counter at which the flow's class is inferred using PL and FL features is called as inference point [4]. A packet counter value of a flow denotes its phase.

ISBN 978-3-903176-63-8 © 2024 IFIP

implement a traffic-specific eviction policy that decides which flows to prioritize and which flows to evict upon a collision (more details in §IV).

Profiler. Aggregated ML model and PrioritySketch algorithm should be configured with traffic-specific parameters (*e.g.,* thresholds, optimal hash table size, important PL + FL features). Manually identifying optimal parameter values from a large search space is time-consuming and error-prone. We design and implement a profiler that automatically derives optimal configuration to maximize accuracy gains and minimize switch memory usage (§V).

Evaluation. We evaluate `AdaFlow` prototype[3] for various traffic analysis datasets [17]–[25] and compare its performance with most recent works [2]–[6]. `AdaFlow` improves classification accuracy up to 7% compared to its best competitors NetBeacon [4] and Jewel [5] while maintaining a similar switch memory footprint as Jewel [5] (§VII).

II. AdaFlow system design

A. Design goals

System design should be general (**D1**) enough to support a wide range of traffic analysis tasks for network security while minimizing data plane memory resources (**D2**) and maximizing classification accuracy (**D3**).

B. Overview of `AdaFlow`

Fig. 1 shows `AdaFlow` system workflow. `AdaFlow` has two main components: `AdaFlow` data plane and a Profiler at the controller. The system's workflow has 7 steps. ❶ The profiler automatically derives a traffic-specific profile for a given training trace (dataset) (*e.g.,* CICIDS [17], Covert channel [18], [19]), thus satisfies D1. The derived profile has the details of an optimal configuration that maximizes accuracy (D3) while keeping the memory footprint low (D2). The profile includes dataset-specific hyperparameter configuration, data plane algorithm configuration, and a multi-phase aggregated ML model with important PL and FL features. ❷ The controller configures the switch data plane according to the profile such that the incoming packets (❸) are processed by `AdaFlow` data plane to collect PL and FL features ❹ specified in the profile. ❺ Next, flows are classified using their features as input to a multi-phase aggregated ML model ❻. The inferred class and associated confidence are used by ❼ PrioritySketch to resolve collisions such that important flows are prioritized (*e.g.,* malicious flows for intrusion detection).

C. `AdaFlow` data plane

In this section, we walk you through packet execution paths in the `AdaFlow` 's data plane as shown in Fig. 1. Each path is shown in a different color. `AdaFlow` skips processing those packets that hit rules ($\langle pkt.5t, class \rangle$) in the packet forwarding table (Green path) and applies associated action (*e.g.,* drop, send to the controller, forward). These rules represent the decisions made after classifying a flow using `AdaFlow` . The other three paths are triggered if a flow misses the packet forwarding table.

Blue path. The flow ID storage is a hash table indexed by flow_ID, that is, $pkt.5t$ (packet's 5-tuple). If no collision[4] with the residing flow, then the first `n`−`1` packets of the packet's flow follow the blue path, where `n` is the point at which the inference is switched to include FL features in addition to PL features. For these packets, `AdaFlow` extracts PL features and checks whether timeout[5] happened or RST/FIN flag is set. If the condition is met, the packet follows the solid line: fetches FL features, resets to PL features, infers its flow class from the ML model (more details in §III), and the `flow_ID` of malicious flows are sent to the controller as a digest message. Upon receiving the digest, the controller installs user-defined rules (via FU) (e.g., which action to take against the subsequent packets with the same $pkt.5t$) in the packet forwarding table. For malicious packets, the action could be drop or send to a specific port for further analysis.

If the condition is not met, the path followed is shown in a dotted line; FL features are updated and the current pkt is classified using PL features until the packet count reaches n. Finally, `AdaFlow` applies user-defined class-specific packet processing rules, that is, if the packet's class is determined as malicious either drop or send to the controller, otherwise skip, and forward to the next module in the data plane.

Orange path. When no collision is detected and the number of packets of a flow exceeds `n`, the path is similar to the solid blue path when the timeout condition is met. Otherwise (dotted orange line), `AdaFlow` updates FL features and infers the flow class using the ML model. If a flow's class prediction confidence crosses a determination threshold ($P(y) > dt$), the flow's priority flag in the flow priority storage is set to 0' (initially all are 1s). The updated priority enables collision handling in the PrioritySketch algorithm such that high accuracy (D3) and low memory overhead (D2) design goals are met (more details in §IV). In addition, `AdaFlow` data plane shares flow_ID and class_ID with the controller and applies user-defined class-specific packet processing rules as mentioned earlier.

Red path. If an incoming packet *collides* with a residing flow, the execution path depends on the residing flow's `priority` in the flow priority storage; `priority` = 1 (solid line) means the residing flow is not classified (i.e., not yet reached the determination threshold) and, thus prioritized as it does not contain reliable FL features. In this case, the incoming pkt is classified based on PL features. Whereas if the residing flow's

[3]The authors have provided public access to their code at https://github.com/networked-systems-iith/AdaFlow.

[4]collision occurs when the index at which a residing flow is stored does not match with $h(pkt.5t)$, where $h(.)$ is a hash function.

[5]We consider two timeout thresholds: idle (δ_{idle}) and active (δ_{active}). Idle timeout occurs when the difference between the current pkt timestamp and the last seen packet timestamp of the residing flow exceeds δ_{idle}. Active timeout occurs when the residing flow duration exceeds δ_{active}. The rationale for using active timeout is to evict persistent flows to achieve low resource usage and high real-time traffic classification.

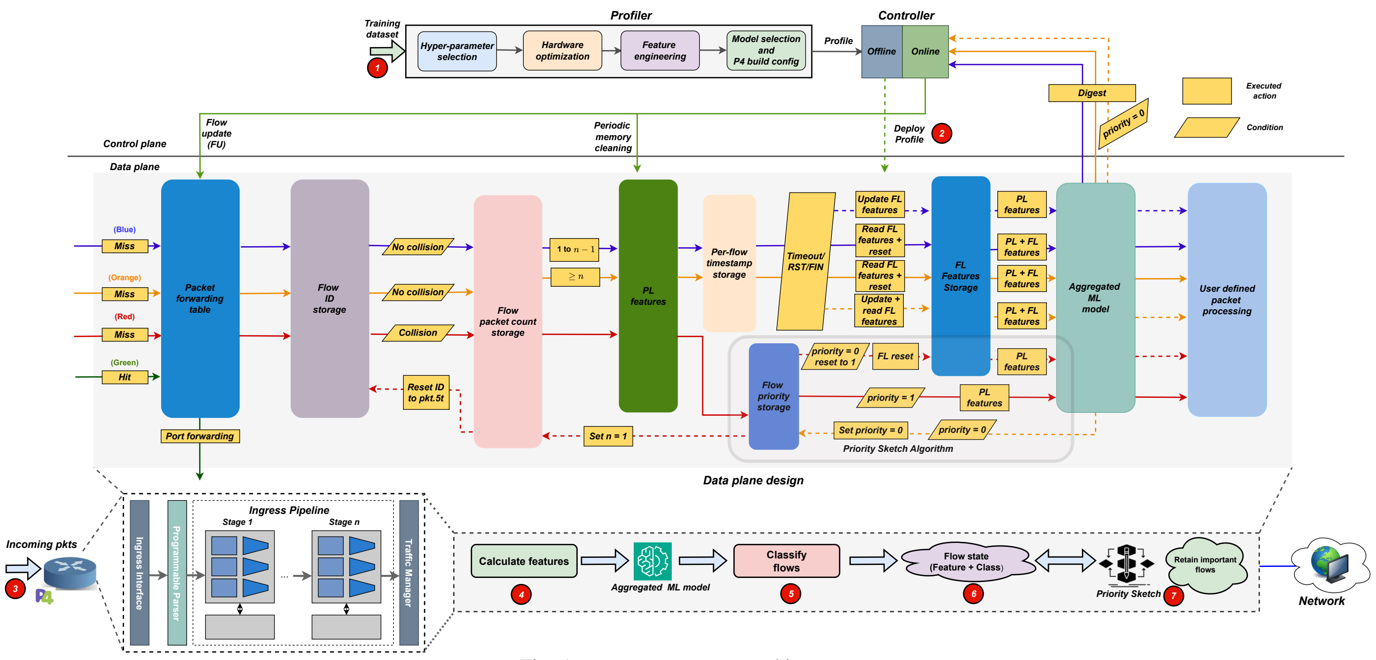

Fig. 1: AdaFlow system architecture

priority = 0, it means that the respective FL features are sufficient enough to determine its class, thus the residing flow can be replaced with the incoming packet's flow. Accordingly, the priority is reset to 1, the FL features are re-initialized with the incoming packet's PL features, and the user-defined class-specific packet processing rules are applied. More details of the PrioritySketch algorithm entailed by this path are covered in §IV.

Controller. Apart from updating the packet forwarding table upon receiving digests (FU), the controller also periodically clears old entries in the packet forwarding table to avoid overflow and clears the storage (i.e., register entries) in the data plane.

Analysis. The rate at which the data plane forwards digests (contains flow_ID and class_ID) to the controller is proportional to the sum of the flow completion rate, flow timeout rate, and the rate at which flows reach their determination threshold.

Implementation on PISA-based programmable switches like Intel Tofino [26]. AdaFlow 's data plane is carefully designed considering the hardware constraints of programmable switches. Our implementation complies with the workflow Fig. 1. The position of storage resembles their implementation using read/write on registers in the multi-stage pipeline architecture like in PISA. The back arrow paths resemble the resubmit mirrored *pkt* operation in the pipeline. Due to the page limit, we omit the details of our implementation on Tofino target.

In the next section, we present the design of our aggregated ML model (§III) and the workings of the PrioritySketch algorithm (§IV) during collisions (red path in the workflow).

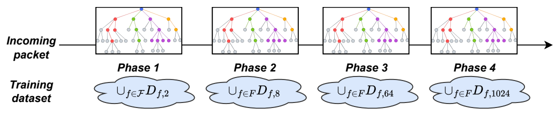

Fig. 2: Sequential model: model M_i for i^{th} phase is trained on $\cup_{f\in F} D_{f,i}$, where $D_{f,i}$ corresponds to a set of flow features and the corresponding labels for a flow f at i^{th} phase. Strategy followed by NetBeacon [4].

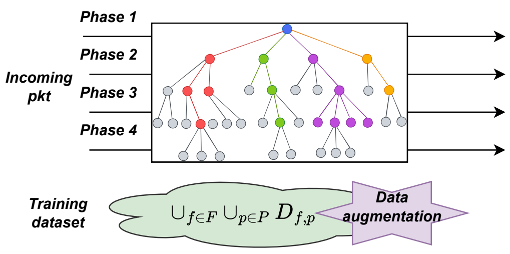

Fig. 3: Aggregated model. Strategy followed by AdaFlow .

III. Multi-phase Aggregated ML Model

Packets in a flow are located at different phases[6], so each phase has different FL features [4]. Using PL and FL features for inference, the ML model should be able to efficiently classify flows at *multiple phases* (should have many inference points[7]). Also, the PL features can be used for inference when FL inference cannot be performed due to collisions. However, this requires separate ML models to be deployed for PL features and PL+FL features which have a high switch memory overhead.

The key idea of our approach is to have a *single ML model* that (i) relies on PL features for classifying early packets of a flow (<n) while updating FL features in parallel, and (ii)

[6]A phase of a flow depends on the number of packets processed so far, thus a flow can have as many phases as the number of packets. FL features vary as the flow moves from one phase to another [4].

[7]Phases at which classification is performed and class is inferred are called inference points.

Algorithm 1 Aggregated ML model training

```
Input: PCAP trace, trigger packet point $n$, set $I_S$ for storing dense inference points to perform data augmentation, timeouts $\delta_{idle}$ and $\delta_{active}$, $D$ denoting set of dataset sample points.
Variables: $PktCount$ denoting number of packets corresponding to a 5-tuple, $FeaturesFL$ denoting FL feature, $FeaturesPL$ denoting PL feature, $S$ denoting a dataset sample, $LastTime$ and $FirstTime$ denoting latest and first timestamp of $pkts$ in a flow.
Output: Decision tree-based ML model
 1: for $pkt$ in trace do
    // Obtain $PktCount$ and check if there is timeout or RST/FIN flag set
 2:     $ID \leftarrow pkt.5t$
 3:     $PktCount_{stored} \leftarrow$ IncrementRead($PktCount$, $ID$)
 4:     $FirstTime_{stored} \leftarrow$ Read($FirstTime$, $ID$)        ▷ Action(Reg, Idx, Value)
 5:     if $FirstTime_{stored} == 0$ then
 6:         Update($FirstTime$, $ID$, $0$)
 7:     Update($LastTime$, $ID$, $pkt.time$)
 8:     $FirstTime_{stored} \leftarrow$ Read($FirstTime$, $ID$)
 9:     $LastTime_{stored} \leftarrow$ Read($LastTime$, $ID$)
10:     activeTimeout $\leftarrow$ $LastTime_{stored}$ - $FirstTime_{stored} > \delta_{active}$
11:     idleTimeout $\leftarrow$ $pkt.time$ - $FirstTime_{stored} > \delta_{idle}$
12:     $timeout \leftarrow$ activeTimeout $or$ idleTimeout
13:     $flag \leftarrow pkt.RST$ or $pkt.FIN$
14:     if $timeout$ or $flag$ then
    // If timeout/flag, prepare samples using PL+FL features, and reset features and timestamps. Sample weight depends on PktCount.
15:         $FeaturesFL_{stored} \leftarrow$ Read($FeaturesFL$, $ID$)
16:         $S \leftarrow \langle pkt.5t, FeaturesPL, FeaturesFL_{stored}, PktClass \rangle$
17:         $w(S) \leftarrow max(1, ne^{\frac{PktCount_{stored}-1}{n}-1})$
18:         $D \leftarrow D \cup S$
19:         Reset($PktCount$, $ID$, $1$)
20:         Reset($FeaturesFL$, $ID$, $FeaturesPL$)
21:         Reset($FirstTime$, $ID$, $pkt.time$)
22:         Reset($LastTime$, $ID$, $pkt.time$)
23:     else if $PktCount_{stored} < n$ then
    // Early samples of a flow are prepared only using PL features.
24:         $FeaturesFL_{stored} \leftarrow \phi$        ▷ $\phi$ is unattainable by FL features
25:         $S \leftarrow \langle pkt.5t, FeaturesPL, FeaturesFL_{stored}, PktClass \rangle$
26:         $w(S) \leftarrow 1$
27:         $D \leftarrow D \cup S$
28:         Update($FeaturesFL$, $ID$, $FeaturesPL$)        ▷ Update FL feats
29:     else if $PktCount_{stored} \in I_S$ then
    // $I_S$ denotes set of samples added at dense packet locations via data augmentation. Samples formed using PL+FL features.
30:         Update($FeaturesFL$, $ID$, $FeaturesPL$)        ▷ Update FL feats
31:         $FeaturesFL_{stored} \leftarrow$ Read($FeaturesFL$, $ID$)
32:         $S \leftarrow \langle pkt.5t, FeaturesPL, FeaturesFL_{stored}, PktClass \rangle$
33:         $w(S) \leftarrow ne^{\frac{PktCount_{stored}}{n}-1}$
34:         $D \leftarrow D \cup S$
35:     else        ▷ Update FL feats
36:         Update($FeaturesFL$, $ID$, $FeaturesPL$)
37: Decision tree based ML model trained on $D_{train} \subset D$ and inference rules derived. Finally, inference is done on $D_{test} = D - D_{train}$.
```

once inference is switched to include both PL + FL features (for packets ≥n), the model *accurately* monitors *most* of the phases in a flow (see Fig. 3). To realize this idea, we carefully use *data augmentation* while training the ML model. More specifically, we augment the dataset to capture information of a flow for a *set of phases that are very closely spaced*. For instance, the training dataset could be augmented to capture flow information at {2nd, 5th, 10th, 15th, 20th, ...} packet. The model is trained on $\cup_{f \in F} \cup_{p \in P} D_{f,p}$, where P corresponds to a set of very closely spaced phases, D denotes training set, and F denotes set of flows keyed by 5-tuple. In this way, we can come up with a single model to be deployed in the data plane while most of the (dense) phases are monitored accurately. Thus, our design is memory efficient (D2) and more accurate (D3) compared to the sequential model Fig. 2 (details in §VI).

Training algorithm. We show ML model training on a dataset prepared from PCAP traces in Alg. 1. In lines 1 to 22, we keep track of the number of packets in a flow based on $pkt.5t$ and also whether the flow is under a timeout or if there is a FIN/RST flag. If the condition is met, the training sample S is collected in D and FL features are reset. Next (lines 23-28), if timeout or RST/FIN flag condition is not met and if the number of $pkts$ in a flow $PktCount < \mathtt{n}$ (where n is the packet point at which we decide whether to switch to use FL features in addition to PL features), we add a sample S containing only PL features while assigning values unattainable by FL features (ϕ) to the dataset D. Meanwhile, FL features are updated for that flow but the sample is not prepared by using them. The sample created using PL features is assigned a training weight of 1. In lines 29-36, sample S with both PL and FL features is added to the dataset D at every phase denoted by $PktCount \in I_S$, where I_S contains a set of very closely spaced phases or packet points. Sample S is assigned training weight $ne^{\frac{PktCount}{n}-1}$ (works well for diversified use-cases[8]) whenever for that flow sample $PktCount \geq n$ (higher weights to flows having more packets). Finally (line 37), after D is formed, the ML model is trained on D_{train}, and inference rules are derived and pushed to the data plane. Then the design is tested on D_{test}, where $D = D_{train} \cup D_{test}$.

In summary, our ML model design is a single aggregated model that can capture the essence of both PL and FL features by processing PL features in flow samples for early packets ($PktCount < n$) and switching to flow samples with both PL and FL information when $PktCount \geq n$. The model is trained on flow samples (I_S) containing both PL+FL information at closely spaced phases via data augmentation.

Additional details on training. We augment the training dataset to obtain PL+FL features at packets: {n, n+Δ, n+2Δ, ..., n+kΔ} ∪ 2^i, where $2^i \leq \mathtt{n} + k\Delta$. We obtain n, Δ and k via profiler in §V. The search space of Δ should be such that dense inference points are covered.

Implementation. A decision tree (DT) is trivial to implement as there's only one tree [3], [4] which can be converted to a set of inference rules. To implement random forest, we use individual feature tables, code tables (tree traversal paths), inference class table, and majority voting tables [3]. For xgboost, we decide the class by aggregating the confidence score [4] (for each root-to-leaf paths) via *sigmoid(.)*. Due to the page limit, we keep the implementation details short.

Using ML model at runtime. Once the model is pushed to the data plane, we classify early packets of flow ($< n$) using PL features, and we classify flows at every packet $\geq n$ using PL+FL features[9] (Fig. 1) until the confidence threshold is reached *i.e.,* $P(y) > dt$ or one of the eviction conditions are met (timeout or RST/FIN flag).

[8]Terms datasets and use-cases have been used interchangeably.

[9]Some FL features (*e.g.,* per-flow average or variance of pkt size) cannot be obtained for all phases (they involve division). For such features, `AdaFlow` calculates them for a few 2^i inference points using bitwise shift like in [4].

IV. PrioritySketch algorithm

We design PrioritySketch algorithm runs in the data plane to resolve collisions efficiently and achieve high accuracy while keeping the memory footprint low. One strawman solution to resolve collisions is to simply evict the flows based on three eviction conditions: idle timeout, active timeout, or flag-based eviction. Upon collision with an incoming flow, if the residing flow meets any one of the conditions, it is evicted and replaced with the incoming flow. Otherwise, we ignore incoming flow and simply update the flow features of the residing flow. This approach is equivalent to giving equal priority to both flows, thus random and may not prioritize important flows affecting accuracy. A high collision rate with less memory would make it even worse, thus having a further impact on accuracy.

We observe that for CIC-IDS2017 [17], [27] PCAP traces, the collision percentage is 38% to 52% with 16K hash table entries. With equal priorities to all flows, we may not be able to exploit dataset-specific flow characteristics which could improve accuracy. For example, for CICIDS Tuesday trace, there are few malicious flows compared to benign flows and most of the flows are short. In such an environment, upon collision detection with a residing flow, giving higher weightage to malicious flows over benign flows would improve accuracy while still using less memory. Based on this idea, we propose PrioritySketch algorithm which proactively identifies potential flows that we wish to prioritize and keep their state in the memory when an incoming flow collides with them.

Offline priority assignment. The parameter *"priority"* controls the flow retainability by assigning priority offline. As mentioned earlier, a decision tree-based aggregated ML model is deployed in the data plane in the form of inference rules. One can visualize each rule r as a set of PL and FL feature ranges and tagged with two values: an inference class $y(r)$ and confidence $P(y|r) \in [0, 1]$. A flow class is determined only when $P(y) > dt$, where dt is the determination threshold. This means flows with higher dt are classified with more confidence.

We observe that an eviction policy based on a single determination threshold for both malicious flows and benign flows would not give good classification accuracy across datasets due to their difference in traffic characteristics such as the percentage of long flows and the percentage of malicious flows. Therefore, we consider two thresholds, dt_M and dt_B for malicious and benign flows respectively, then we tag each rule r with priority value as follows,

$$priority(r) = \begin{cases} 1 - \mathbb{1}\{\mathbb{P}(y = mal|r) > dt_M\} & y(r) = mal \\ 1 - \mathbb{1}\{\mathbb{P}(y = ben|r) > dt_B\} & y(r) = ben \end{cases}$$

If we assign different priorities to each of the flow classes,

$$priority(r) = 1 - \mathbb{1}\{\mathbb{P}(y = c|r) > dt_c\} \quad y(r) = c$$

where $c \in [k]$ is a class label associated with rule r in the k-class multi-classification problem. After all inference rules are tagged with appropriate `priority` (either 0 or 1), we update the ML model in the switch data plane.

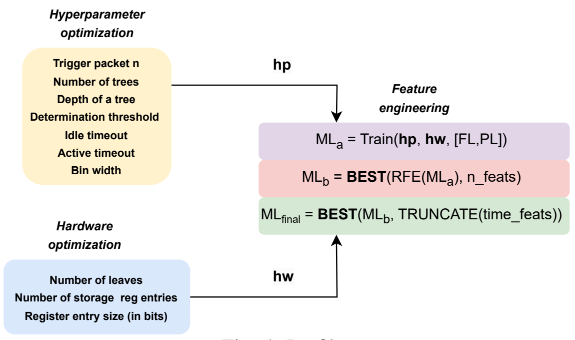
Hyperparameter optimization
Trigger packet n
Number of trees
Depth of a tree
Determination threshold
Idle timeout
Active timeout
Bin width
hp
Feature engineering
ML_a = Train(hp, hw, [FL,PL])
ML_b = BEST(RFE(ML_a), n_feats)
ML_final = BEST(ML_b, TRUNCATE(time_feats))
Hardware optimization
Number of leaves
Number of storage reg entries
Register entry size (in bits)
hw

Fig. 4: Profiler

Using priority at runtime. At runtime, when a flow exceeds the determination threshold and is mapped to an inference rule (r), the corresponding *priority(r)* value and the flow class are retrieved. Also, the flow's priority flag is updated in flow priority storage as shown in Fig. 1. For instance, when the majority of the flows are short and only a few of them are malicious, high dt_M (say 0.9) and low dt_B (say 0.1) would give good recall; since the majority of cached flows are benign and benign flows are quickly evicted (*i.e., priority=0*) as their confidence exceeds determination threshold much faster than malicious flows, thus creates space for new flows. Whereas the prioritized residing malicious flows (*i.e.,* priority=1) most likely collide with incoming benign flows, thus preserving recall. We present more details on the algorithm's performance for different determination thresholds across datasets in §VII.

V. Profiler

The complete workflow of the profiler component of `AdaFlow` is shown in Fig. 4. Hyperparameter optimization and hardware optimization blocks show a bunch of parameters we select from the space of configuration (some even mentioned in [4], [5]). We explain the workflow by connecting to Fig. 4 as follows:

1) For each configuration [**hp, hw**], we train an ML model (following Alg. 1), ML_a using the complete FL and PL features from a dataset.
2) Next, we perform recursive feature elimination [28] by iterating over the number of features derived using ML_a and obtain the best model ML_b. This helps us arrive at not only the optimum number of features but also those that are most important.
3) Next, we try to fit the time-based features time_feats in the hardware memory by adopting the truncation strategy used by [3] and get the best of all the candidate models.
4) Finally, we repeat steps 1-3 over the entire [**hp, hw**] search space to obtain the best candidate model ML_{final} to be deployed on the target switch.

To select the best model, we maximize $reward = \alpha$.F1-score + $(1 - \alpha).(1 - \rho)$, where ρ denotes fraction of total memory in target switch and α[10] $\in [0, 1]$. The inference rules derived from ML_{final} are then pushed into `AdaFlow` data plane. Note that the ML_{final} model is selected based on final

[10]In our experiments, we keep $\alpha = 0.5$ to maintain balance.

System	Accuracy				Scalability (Memory efficient)	Generality
	Features	Inference points	Training weights	Flow prioritization		
Mousika [2]	PL	✗	✗	✗	✗	✗
Planter [1], [6]	PL	✗	✗	✗	✓	✗
Flowrest [3]	FL	Single	✗	✗	✓	✓
NetBeacon [4]	PL+FL	Sparse	$w(PL) = w(FL)$	✗	✗	✓
Jewel [5]	PL+FL	Single	$w(PL) < w(FL)$	✗	✓	✓
`AdaFlow`	PL+FL	Dense	$w(PL) < w(FL)$	✓	✓	✓

TABLE I: Comparision of `AdaFlow` with related work. NetBeacon uses 4-10 inference points while Jewel uses only 1 inference point. Jewel however prioritizes PL+FL inference over PL-only inference, unlike NetBeacon which assigns the same weights to all samples. Moreover, NetBeacon uses multiple ML models. `AdaFlow` solves issues of both Jewel and NetBeacon using its efficient ML model design in §III and PrioritySketch algorithm in §IV.

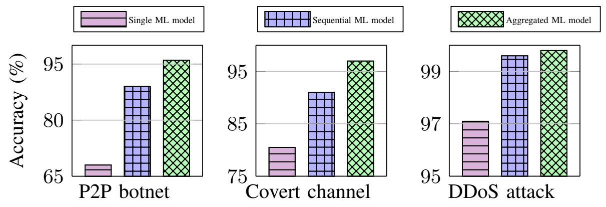

Fig. 5: `AdaFlow` 's aggregated model-based classifier outperforms NetBeacon's sequential model-based classifier by 1.5-11%, and outperforms Flowrest's single inference-based classifier by 3-24%.

FL + PL features FL_{final} + PL_{final} as well as final configurations $\mathbf{hw}_{\text{final}}$ and $\mathbf{hp}_{\text{final}}$.

VI. Comparision with recent work

PL features or FL features. Mousika [2] and Planter [6], both rely only on PL features for inference tasks. [4] shows that without FL features, accuracy is poor for multiple use cases. On the other hand, Flowrest [3] considers only FL features for inference tasks but ignoring early packets affects accuracy. Also, it uses a *single inference point* `n` to fix the inferred class which does not work well for datasets with a mix of short and long flows [20].

PL features and FL features. NetBeacon [4] relies on both features but their data plane module consumes more memory as it deploys multiple ML models (one based on PL and FL features, another based on PL features, and another to determine short or long flows). Also, NetBeacon uses few inference points which affects accuracy. For instance, if the inference is done at {2nd, 4th, 8th, 32nd, 256th, 512th, 2048th} packet, the inference of the 2047th packet and the 512th packet are the same, thus fails to monitor 1535 phases in the middle. Jewel [5] proposes a unified ML model considering both features; though this approach is memory efficient it has a single inference point, thus hurting accuracy. Moreover, both works do not have collision handling mechanisms. Whereas `AdaFlow` achieves memory efficiency by carefully designing the data plane module with a single aggregated model, resolves collisions based on priority, and evicts flows at the right time. In parallel, `AdaFlow` improves accuracy by using dense inference points and by enabling dataset-specific determination threshold configuration. We summarize the comparisons in Table I. From Fig. 5, we observe `AdaFlow`'s aggregated model-based classifier outperforms NetBeacon's sequential PL+FL model accuracy by 1.5-11% and reduces memory usage by 15.68-78.39% (see §VII-C). More details on the experiment setting are in §VII-B.

VII. Evaluation

We evaluate `AdaFlow` to study its performance in terms of classification accuracy and switch resource overheads. In summary, compared to the state-of-the-art works, `AdaFlow` improves F1 score consistently for various datasets while keeping memory footprint low and minimal-to-zero impact on packet-processing throughput.

A. Traces and metrics

Evaluation traces. We evaluate `AdaFlow` on four tasks[11]:

- **Covert channel detection.** The task identifies Covert channel traffic encoded by two censorship resistance tools, Facet [18] and DeltaShaper [19], from benign skype traffic.
- **P2P botnet detection.** This task uses the PeerRush dataset [20] which comprises flows produced by four benign P2P applications (uTorrent, Vuze, Frostwire, and eMule), and two P2P botnets (Waledac and Storm).
- **DDoS attack detection.** This task uses CIC-DDoS2019 dataset [21] which consists of a mix of benign traffic and various real-time DDoS attacks (MSSQL, SSDP, CharGen, NTP, TFTP, SYN Flood, UDP Flood, and UDP Lag).
- **Intrusion detection dataset.** This task uses *fixed version* of CIC-IDS2017 dataset [17], [27]. This dataset consists of 5 IP traces: Monday (benign traffic), Tuesday (FTP and SSH Patator), Wednesday (DoS and Heartbleed), Thursday (Infiltration and Web-Attacks), and Friday (DDoS, Portscan and Botnets).

Metrics. To understand `AdaFlow`'s *classification performance*, we consider per-packet[12] based F1 score[13] given by $\frac{2TP}{2TP+FP+FN}$. To understand switch overheads, we measure memory utilization and packet recirculation overhead.

B. Classification Performance

This section presents the quantitative benefits of `AdaFlow` 's aggregated ML model and the PrioritySketch algorithm running in the data plane via simulations and experiments on a real testbed with a Tofino switch.

1) Simulation results.

We simulate `AdaFlow` on a machine with 40-core, 2 x Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, and 256GB DDR4 memory. We have two variants of `AdaFlow` : $dt_M = dt_B = 0.8$ and $dt = [dt_M, dt_B]$ optimized as per the profiler called as $\texttt{AdaFlow}_{Optimized}$. We compare `AdaFlow` variants with NetBeacon ($dt = 0.8$), Flowrest, and Jewel. For all the systems, we use optimized parameters derived using the profiler with hash table entries set to 4096 for the covert

[11]We additionally introduce UNIBS [22], UNSW [23], ToN-IoT [24], and IoT23 [25] in §VII-B2

[12]Metric is calculated on per-packet basis. For instance, if a flow consisting of N packets is classified, then all those N packets share the same class. Any flow containing N $pkts$ that is unclassified (can happen in Flowrest [3] when the flow is overwritten) due to collision is assumed to be incorrectly classified (as in all such N packets being incorrectly classified).

[13]We consider F1 score averaged across all classes.

Tasks	P2P app fgpt.	Covert channel (Facet)	Covert channel (Deltashaper)	DDoS Detection	CIC-IDS (Tuesday)	CIC-IDS (Thursday)
ML model	Random Forest	XGBoost	XGBoost	XGBoost	Random Forest	Random Forest
Training flows	4,446,709	28,155	11,657	47,752	595,358	617,030
Testing flows	423,452	9034	3932	191,009	129,667	142,344
Class ratio (ben/mal)	10:1	1:1	1:1	6:1	100:3	1000:5
No. of inference rules	512	256	256	32	256	256
δ_{idle} , δ_{active}	$60s$, $3600s$	$5s$, $30s$	$5s$, $30s$	$256ns$, $2s$	$5s$, $120s$	$5s$, $120s$

TABLE II: Experimental setting for simulations

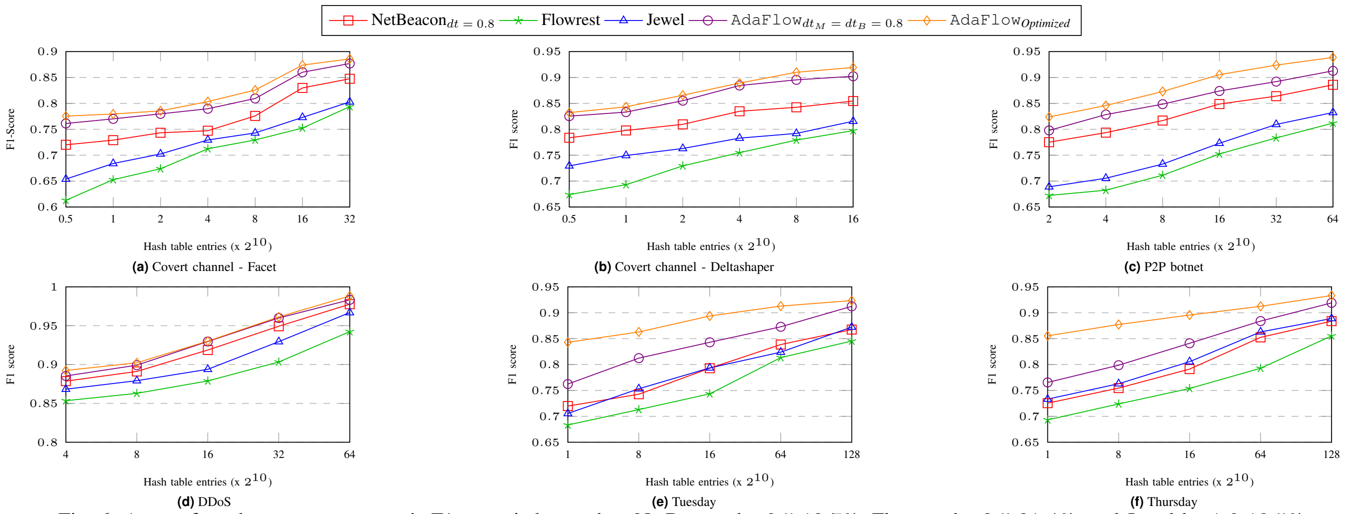

Fig. 6: Across four datasets, AdaFlow 's F1 score is better than NetBeacon by 0.5-12.7%, Flowrest by 2.5-21.4%, and Jewel by 1.0-18.5%.

channel task and 65, 536 for the other 5 tasks. This is done for fair comparison. Table II summarizes the AdaFlow 's settings for experiments across tasks.

Covert channel. Fig. 6a and 6b show the F1 score for delta shaper and facet tasks, respectively. AdaFlow with $dt_M = dt_B = 0.8$ outperforms NetBeacon with $dt = 0.8$ by 3-5%, Flowrest by 5-10%, and Jewel by 4-7%. Since $dt_M = dt_B$, dataset-specific flow prioritization does not influence accuracy, thus the benefits are mainly due to the AdaFlow's aggregated ML model design. AdaFlow's optimized version leverages PrioritySketch 's dataset-specific flow prioritization by enforcing different determination thresholds, thus giving an additional gain of 0-3%. The optimized version uses high dt values ($dt_M = 0.93$ and $dt_B = 0.84$). Since most of the flows are long flows with malicious and benign flows in equal proportion, both types of flows would participate in storage collisions. However, the collision rate is very low for this task due to less number of concurrent flows (Table II). Thus it is preferred to keep both dt_M and dt_B high for this task; otherwise, low dt_M or dt_B would lead to poor accuracy due to premature flow classification.

P2P botnet detection. From Fig. 6c, we observe that the F1 score of AdaFlow with $dt_M = dt_B = 0.8$ outperforms NetBeacon with $dt = 0.8$ by 2-6%, Flowrest by 6-10% and, Jewel by 4-8%. This is mainly due to the AdaFlow's aggregated ML model design as explained earlier. AdaFlow's optimized version gives additional gains of 2.5-5%. This task has a mix of short and long flows where the majority are benign flows. For flows with duration $> 5s$ (about 23%), we set $dt_M = 0.93$ and $dt_B = 0.82$, and for flows with duration $\leq 5s$ (about 77%), we set $dt_M = 0.78$ and $dt_B = 0.66$. This is based on the observation that flows with less duration are evicted sooner and hence their confidence exceeds the threshold faster than flows with more duration. Lower dt for such flows therefore does not affect accuracy. Flows with more duration are less in number and have higher dt for more accurate classification. We keep $dt_M > dt_B$ since there are much fewer malicious flows than benign flows, so most collisions occur due to benign flows.

DDoS attack detection. AdaFlow with $dt_M = dt_B = 0.8$ outperforms NetBeacon with $dt = 0.8$ by 0-2%, Flowrest by 2-4%, and Jewel by 1-2% (Fig. 6d). Whereas optimized AdaFlow's version with $dt_M = 0.92$ and $dt_B = 0.89$ gives additional gains of 0.5-1%. This is because most flows are classified accurately before a collision happens (accurate early classification), thus high dt_M and dt_B values are preferred.

Intrusion detection for Tuesday and Thursday datasets. From Fig. 6e and 6f, we observe AdaFlow with $dt_M = dt_B = 0.8$ outperforms NetBeacon with $dt = 0.8$ by 6-10%, Flowrest by 7-12% and Jewel by 6-11%. The optimized AdaFlow version with PrioritySketch gives additional gains of 2-11%. This version uses $dt_M = 1$ and $dt_B = 0.37$ for Thursday trace, and $dt_M = 0.95$ and $dt_B = 0.24$ for Tuesday trace. Low dt_B and high dt_M values are preferred for the following reasons. The majority of the flows are short and only a few of them are malicious. High dt_M and low dt_B would give good recall; since the majority of cached flows are benign, benign flows are quickly evicted as their confidence exceeds the determination threshold much faster than malicious flows. Whereas the prioritized malicious flows may most likely collide with incoming benign flows, thus preserving recall while maintaining precision. Further, since the majority of flows are short, once a residing flow is classified as malicious or benign (based on confidence), it remains the same until

	Mousika	Planter	Flowrest	NetBeacon	Jewel	AdaFlow
Covert	55.21%	58.78%	82.56%	87.88%	84.55%	**93.63%**
P2P fgpt	76.22%	77.74%	84.44%	92.27%	86.73%	**95.46%**
DDoS	83.49%	89.34%	96.59%	99.35%	99.27%	**99.57%**
Tuesday	72.43%	75.54%	85.53%	90.32%	89.05%	**95.34%**
UNIBS	90.35%	91.56%	96.40%	97.62%	98.35%	**99.24%**
UNSW	82.01%	79.85%	80.69%	84.58%	87.32%	**93.57%**
ToN-IoT	27.55%	70.49%	73.46%	78.05%	75.70%	**85.22%**
IoT23	86.05%	88.14%	82.85%	91.08%	91.31%	**93.44%**

TABLE III: F1 score for various datasets: `AdaFlow` gives up to 7% better F1 score than the second best system. As for the second best system, Jewel takes the lead in UNIBS, UNSW and IoT23 by 0.2-3% while NetBeacon leads in the rest of datasets by 0.1-6%. We use xgb-distilled (covert and DDoS) and RF-distilled BDTs (rest of datasets) in Mousika [2].

its confidence exceeds the respective determination threshold. Therefore, early eviction of benign flows due to low dt_B would not have much impact on precision. We get similar results for Wednesday and Friday trace. Both the traces have equal mix of benign and malicious flows, and most of the flows are short. Thus, keeping both dt_B and dt_M as high is preferred.

2) Testbed results.

We conduct testbed experiments on 3.2 Tbps 12-stage Tofino 1 ASIC. We replay PCAP traces via `tcpreplay`. Further, we also inject benign background traffic from the CIC-IDS2017 Monday trace (as non-target traffic) at 40 Gbps speed (besides use-case traffic). This background traffic is not a target for inference, hence this traffic is bypassed by the packet forwarding table (as shown in Fig. 1). We demonstrate that the presence of non-target traffic does not hinder the working of `AdaFlow`. We compare the `AdaFlow` 's optimized version with each of the best versions of systems mentioned in Table III. For each task, the same type of ML model is used across systems (XGboost for Covert and DDoS, and random forest for the rest). We observe that `AdaFlow` outperforms the best competitor by $0.4 - 7\%$.

Across tasks, the second-best system changes between NetBeacon and Jewel[14] (multiple sparse inference points vs prioritizing FL+PL inference, see Table I). In detail, Jewel is found to work well for datasets containing short flows where a single reasonable inference might work well. Especially, when there are high storage collisions, NetBeacon will use PL-only inference for short flows while Jewel will prioritize PL+FL inference leading to better accuracies. However, if a dataset has diverse flow characteristics (*e.g.*, a mix of short and long flows), then using a single inference point might not lead to good detection performance. It requires multiple inference points for an accurate detection of diverse flows. In this case, NetBeacon surpasses Jewel. In summary, `AdaFlow` surpasses both Jewel and NetBeacon consistently across multiple datasets due to ML model design (§III) and PrioritySketch algorithm (§IV).

C. Switch overheads

Memory and compute overheads. We develop dataset-specific prototypes of three systems (`AdaFlow` 's optimized version, Jewel, and NetBeacon) and deploy them on a Tofino switch. Fig. 7 shows the resource overheads. Compared to NetBeacon, `AdaFlow` consumes fewer resources (18.92-78.39% TCAM, 32.67-52.45% SRAM, 15.68-44.6% sALUs, 52.89-60.14% VLIWs) while being at par with Jewel. Improvements over NetBeacon are mainly due to our single aggregated ML model for PL and FL features (instead of multiple individual models) and using natively supported range matching primitive (instead of customized *range marking* primitive which consumes more TCAM and SRAM resources).

Packet recirculation overhead. We measure overheads of packet recirculation rate in terms of its impact on packet processing throughput. We observe there is not much difference in the throughput across the three systems. On a 40 Gbps link, `AdaFlow` with $dt_M = dt_B = 0.8$ reported 39.63 Gbps, 39.65 Gbps, and 39.64 Gbps for P2P botnet, Covert channel, and DDoS attack datasets, respectively. Whereas NetBeacon reported 39.64 Gbps, 39.65 Gbps, and 39.67 Gbps, and Jewel reported 39.63 Gbps, 39.66 Gbps, and 39.63 Gbps. Average per-packet latency of `AdaFlow` across 7 datasets (Fig. 7) is 356 - 617 nanoseconds, confirming a sub-microsecond delay for in-switch classification.

VIII. Related Work

Classification at the control plane. Works [7], [29] in this category extract FL features in the data plane but offloads traffic classification to the control plane for fine-grained analysis. They either use statistical analysis methods [29] or supervised ML-based classifiers [7]. However, unlike `AdaFlow`, they are not designed to achieve line speed (Tbps) classification leading to increased latency and delay in making crucial decisions to the incoming packets (drop, route to network, or route to another module for further analysis).

Using rule-based classifiers. Works in this category use pre-defined traffic filtering policies [30]–[32] to classify traffic in data plane. Such works only deal with specific types of traffic (*e.g.,* DDoS attacks). In contrast, `AdaFlow` deploys supervised decision-tree-based learning models to the data plane which enables it to classify a wide range of network traffic (generality).

Using supervised ML classifiers. Deploying machine learning models directly to the data plane is an emerging research area. Works such as [2]–[6] deploy supervised decision-tree-based ML classifiers (due to their model structure resembling match-action pipeline of PISA based programmable switches) directly in the data plane which enables them to classify a wide variety of network traffic (at line speed) using FL features extracted from the incoming packets. `AdaFlow` falls in this line of work and improves classification accuracy while keeping the data plane overheads low.

IX. Conclusion and Future Work

This paper presents `AdaFlow`, an efficient in-network traffic classifier for various security tasks using a programmable switch. `AdaFlow` is designed to accurately classify traffic with diverse flow characteristics while keeping the data plane memory footprint low. The key idea is to incorporate traffic-specific heuristics while training ML models and resolving

[14]Authors in their experiments assumed NetBeacon used only single tree DT for all datasets while we normalized the ML models comparison (RF and Xgboost) for fairness.

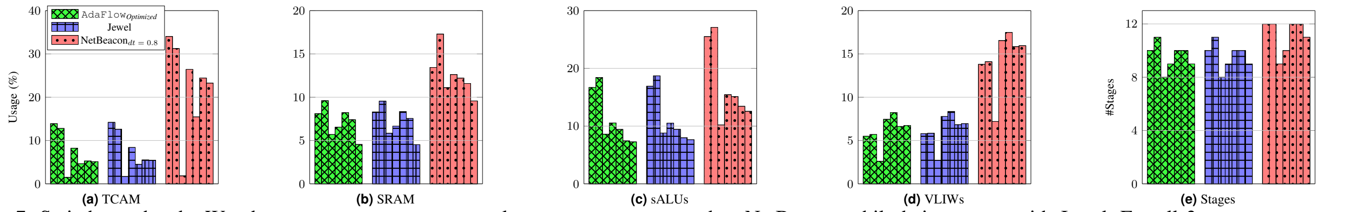

Fig. 7: Switch overheads: We observe `AdaFlow` consumes less resources compared to NetBeacon while being at par with Jewel. For all 3 systems, we use 65,536 hash table entries and 256, 512, 32, 370, 256, 256, and 256 ML model inference rules (deployed in match-action tables) for each of the seven tasks: Covert channel [18], P2P botnet [20], DDoS [21], UNIBS [22], UNSW [23], ToN-IoT [24], and IoT23 [25].

flow-level collisions. Compared to the state-of-the-art work, `AdaFlow` improves accuracy without consuming extra switch resources. As a part of future work, we will evaluate and minimize `AdaFlow` overheads in terms of control- and data plane bandwidth.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their thoughtful feedback and additional insights. We also thank Ranjitha, Harish and Kotaro for giving valuable feedback on the earlier drafts. This work is supported by National Security Council Secretariat (NSCS), India.

REFERENCES

[1] C. Zheng and N. Zilberman, “Planter: seeding trees within switches,” in *Proceedings of the SIGCOMM’21 Poster and Demo Sessions*, 2021.
[2] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, “Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation,” in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, 2022.
[3] A. T.-J. Akem, M. Gucciardo, and M. Fiore, “Flowrest: Practical flow-level inference in programmable switches with random forests,” *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023.
[4] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, “An efficient design of intelligent network data plane,” in *32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association*, 2023.
[5] A. T.-J. Akem, B. Bütün, M. Gucciardo, M. Fiore, *et al.*, “Jewel: Resource-efficient joint packet and flow level inference in programmable switches,” in *IEEE International Conference on Computer Communications*, 2024.
[6] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, “Automating in-network machine learning,” *arXiv preprint arXiv:2205.08824*, 2022.
[7] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. Ramos, and A. Madeira, “Flowlens: Enabling efficient flow classification for ml-based network security applications.,” in *NDSS*, 2021.
[8] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, “Measuring control plane latency in sdn-enabled switches,” in *Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research*, 2015.
[9] 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*, 2017.
[10] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, “Iisy: Practical in-network classification,” *arXiv preprint arXiv:2205.08243*, 2022.
[11] G. Xie, Q. Li, C. Cui, P. Zhu, D. Zhao, W. Shi, Z. Qi, Y. Jiang, and X. Xiao, “Soter: Deep learning enhanced in-network attack detection based on programmable switches,” in *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, 2022.
[12] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, “Inc: In-network classification of botnet propagation at line rate,” in *European Symposium on Research in Computer Security*, 2022.
[13] K. Friday, E. Bou-Harb, and J. Crichigno, “A learning methodology for line-rate ransomware mitigation with p4 switches,” in *International Conference on Network and System Security*, 2022.
[14] X. Zhang, L. Cui, F. P. Tso, and W. Jia, “pheavy: Predicting heavy flows in the programmable data plane,” *IEEE Transactions on Network and Service Management*, 2021.
[15] B. Coelho and A. Schaeffer-Filho, “Backorders: using random forests to detect ddos attacks in programmable data planes,” in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022.
[16] B. Hullár, S. Laki, and A. Gyorgy, “Early identification of peer-to-peer traffic,” in *2011 IEEE International Conference on Communications (ICC)*, 2011.
[17] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward generating a new intrusion detection dataset and intrusion traffic characterization.,” *ICISSp*, 2018.
[18] S. Li, M. Schliep, and N. Hopper, “Facet: Streaming over videoconferencing for censorship circumvention,” *Proceedings of the ACM Conference on Computer and Communications Security*, 2014.
[19] D. Barradas, N. Santos, and L. Rodrigues, “Deltashaper: Enabling unobservable censorship-resistant tcp tunneling over videoconferencing streams,” *Proceedings on Privacy Enhancing Technologies*, 10 2017.
[20] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, “Peerrush,” *J. Inf. Secur. Appl.*, 2014.
[21] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, “Developing realistic distributed denial of service (ddos) attack dataset and taxonomy,” in *2019 International Carnahan Conference on Security Technology (ICCST)*, 2019.
[22] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli, “Detection of encrypted tunnels across network boundaries,” *2008 IEEE International Conference on Communications*, 2008.
[23] 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*, 2019.
[24] G. Bovenzi, G. Aceto, D. Ciuonzo, A. Montieri, V. Persico, and A. Pescapé, “Network anomaly detection methods in iot environments via deep learning: A fair comparison of performance and robustness,” *Comput. Secur.*, 2023.
[25] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, and A. Anwar, “Ton_iot telemetry dataset: A new generation dataset of iot and iiot for data-driven intrusion detection systems,” *IEEE Access*, 2020.
[26] “Barefoot networks, tofino switch,” 2021.
[27] G. Engelen, V. Rimmer, and W. Joosen, “Troubleshooting an intrusion detection dataset: the cicids2017 case study,” in *2021 IEEE Security and Privacy Workshops (SPW)*, 2021.
[28] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, “Gene selection for cancer classification using support vector machines,” *Machine Learning*, 2002.
[29] J. Xing, Q. Kang, and A. Chen, “{NetWarden}: Mitigating network covert channels while preserving performance,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[30] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, “Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches.,” in *USENIX Security Symposium*, 2021.
[31] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, “Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense,” in *ACM SIGCOMM*, (Amsterdam, The Netherlands), August 2022.
[32] 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 *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.