IEEE INFOCOM 2024 - IEEE Conference on Computer Communications | 979-8-3503-8350-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/INFOCOM52122.2024.10621365

Jewel: Resource-Efficient Joint Packet and Flow Level Inference in Programmable Switches

Aristide Tanyi-Jong Akem*†‡, Beyza Bütün*†‡, Michele Gucciardo*, and Marco Fiore*
*IMDEA Networks Institute, Spain, †Universidad Carlos III de Madrid, Spain
{aristide.akem, beyza.butun, michele.gucciardo, marco.fiore}@imdea.org
‡*Equal contributors*

***Abstract*—Embedding machine learning (ML) models in programmable switches realizes the vision of high-throughput and low-latency inference at line rate. Recent works have made breakthroughs in embedding Random Forest (RF) models in switches for either packet-level inference or flow-level inference. The former relies on simple features from packet headers that are simple to implement but limit accuracy in challenging use cases; the latter exploits richer flow features to improve accuracy, but leaves early packets in each flow unclassified. We propose `Jewel`, an in-switch ML model based on a fully joint packet- and flow-level design, which takes the best of both worlds by classifying early flow packets individually and shifting to flow-level inference when possible. Our proposal involves *(i)* a single RF model trained to classify both packets and flows, and *(ii)* hardware-aware model selection and training techniques for resource footprint minimization. We implement `Jewel` in P4 and deploy it in a testbed with Intel Tofino switches, where we run extensive experiments with a variety of real-world use cases. Results reveal how our solution outperforms four state-of-the-art benchmarks, with accuracy gains in the 2.0%–5.3% range.**

I. Introduction

In the next few years, computer networks are expected to support increasingly complex applications with ultra-low latency requirements, including autonomous connected driving [1], augmented or virtual reality [2], or metaverse digital twins [3]. In order to meet the high performance bars set by these services in terms of end-to-end delay, low and possibly guaranteed latency is becoming a core requirement not only for the transport domain, but across all elements that intervene in the communication pipeline. Therefore, such a need for latency optimization also concerns network management decisions that, under today's dominant Software-Defined Networking (SDN) paradigms, are implemented in the control plane, where they realize tasks like traffic classification, anomaly detection, routing optimization, among many others [4]–[6].

With the current push towards network automation, many control-plane network management functions are expected to transition from human-driven to data-driven processes [7]. Specifically, Machine learning (ML) models trained on large amounts of data collected from the network are seen as a prime candidate to implement the intelligence that will enable full network automation [8]: indeed, standard-defining organizations (SDO) are already working on a native integration of ML operations (MLOps) in network management and orchestration (MANO) frameworks [9]. However, the fact that ML operation is bounded to the control plane in SDN and upcoming standards creates a barrier for latency reduction, since it forces a closed-loop communication between the user and control plane with an inherent delay in the order of tens of milliseconds [10]. This prevents ML models from taking very fast decisions and necessarily limits the frequency of their intervention in low-latency application scenarios.

Recent advances in data plane programmability may come to the rescue in these cases. Programmable data planes like Intel Tofino [11] and Netronome Agilio CX Smart Network Interface Cards (SmartNICs) [12], as well as network programming languages like P4 [13] offer opportunities for deploying intelligence in the data plane directly. The approach yields a potential to reduce drastically ML latency to nanosecond-level and to enable line-rate intelligence that operates on each single packet transiting in the network. Yet, it also is extremely challenging to realize in practice: data planes are highly constrained milieux with limited support for mathematical operations, low available memory, and a limited number of supported operations per packet [14]. This makes compute- and memory-intensive model training impractical, and recent efforts have focused on deploying trained ML models in the programmable network hardware for line-rate inference with high throughput and very low latency [15].

Programmable switches have attracted particular attention in this sense. Decision Tree (DT) and Random Forest (RF) models have been successfully implemented in real-world hardware, and have demonstrated the viability of in-switch ML in experiments targeted at malware detection, IoT device identification and traffic classification. These models can be categorized into two large families, depending on whether they perform inference on individual packets [16]–[23], or on flows [24]–[31].

In packet-level (PL) inference, individual packet header fields (*e.g.*, packet length or transport protocol) are used as features. These models are stateless and very lean, as they only need to operate on one packet at a time as in the baseline forwarding tasks the switch is designed for. Yet, they cannot leverage flow-level features, hence can pay a price in terms of accuracy as the problem complexity grows. Conversely, flow-level (FL) solutions implement fairly complex strategies to store stateful per-flow features (*e.g.*, inter-arrival times or maximum packet size , which grants higher performance at the cost of added switch resource utilization.

An overlooked trade-off of PL and FL models is that PL models, although less accurate, can classify *all* packets; instead, more precise FL solutions are forced to delay inference until the moment when flow-level features are reliable, and thus *cannot classify early packets* in each flow [32]. While these early packets are not incorrectly classified by FL models, no action can be taken on them either, and the impact is in the end similar: for instance, in attack detection use cases, the first packets of a malicious flow are let through as benign and may harm the system. The number of early flow packets ignored by FL models vary depending on the model and application use case from 2 to 50 in the existing literature. Yet, an analysis of the traffic in the four measurement datasets we employ in our performance evaluation reveals that: (i) the fraction of flows composed by less than 2 and 50 packets, which are entirely skipped by FL models, is up to 46.44% and within 47.51–99.87%, respectively; and (ii) for flows longer than 2 and 50, early packets that go unclassified by FL models account for up to 67.67% and 98.04% of the total flow length, respectively. Overall, these figures show how FL inference can be oblivious to a non-negligible portion of traffic in realistic use cases, highlighting a problem ignored to date.

In this paper, we close the gap above by proposing `Jewel`, a in-switch ML model that serves as a joint packet and flow level classifier. Our solution enables PL inference on early packets, and shifts to more accurate FL classification as soon as flow-level features are ready. By doing so, `Jewel` takes the best of both worlds: early packets are correctly processed by the in-switch ML model, and at the same time higher accuracy is achieved at flow level.

By developing `Jewel`, we make the following contributions.

Fully joint PL-FL design. An obvious approach to realize the joint PL-FL operation is to deploy multiple ML models in the switch, dedicated to PL or FL inference and triggered on the appropriate packets. In fact, this is the approach adopted by the only solution for PL-FL inference proposed to date, *i.e.*, NetBeacon [33]. However, as we show in our evaluation, this strategy is very expensive in terms of switch resources, which in turn constrains the complexity of the PL and FL models, reducing their performance. Instead, we propose an original ML model design that is *fully joint*, *i.e.*, can perform both PL and FL inference via a single RF, automatically switching between the two modes depending on the packet.

Exhaustive model selection. Our joint PL-FL design allows waiting for a potentially high number of packets to compute more robust flow-level features, while having early packets covered by the PL mode. This is not possible with pure FL approaches that miss all early packets. However, it also enlarges the model design space by a further dimension, *i.e.*, the hyper-parameter indicating at what point of a flow FL inference shall intervene. We thus propose a model selection technique based on an exhaustive search of features and hyper-parameters that explores the full design space and returns models with high accuracy while limiting the model size.

Hardware-native ML training. A joint PL-FL solution naturally tends to be more resource-hungry than the PL or FL models, which calls for improved techniques for minimizing the memory footprint of the deployed ML. We exploit the internal organization of the switch hardware to optimize feature representation and tree structures for fitting in scarce resources like Ternary Content-Addressable Memory (TCAM).

Work	Target		Design			Implementation	
	Packet	Flow	HW-aware design	Model selection	RF support	HW	Open source
Ilsy [16], [17]	✓				✓	✓	
Planter [18], [19]	✓				✓	✓	
Mousika [20]	✓					✓	✓
Henna [21]	✓				✓	✓	✓
NetPixel [22]	✓						✓
Soter [23]	✓					✓	✓
pForest [24]		✓			✓	*	
SwitchTree [25]		✓			✓		✓
NERDS [26]		✓					✓
pHeavy [27]		✓				✓	
BACKORDERS [28]		✓			✓		
INC [29]		✓			**	✓	
Friday et al. [30]		✓	✓		✓	✓	
Flowrest [31]		✓	✓		✓	✓	✓
NetBeacon [33]	✓	✓			**	✓	✓
Jewel	✓	✓	✓	✓	✓	✓	✓

* Partial implementation only. ** Inconclusive based on description and artifacts.

Table I: Summary of existing solutions for in-switch inference via DTs and RFs, compared to `Jewel`. The columns indicate support for (i) PL inference, (ii) FL inference, (iii) ML modeling approach compatible with hardware requirements by design, (iv) systematic and possibly automated model selection, (v) RFs composed of multiple DTs, (vi) implementation in a real-world platform, and (vii) open-source artifacts.

Experimental comparative validation. We implement `Jewel` in P4 into a testbed with production-grade Intel Tofino switches, alongside four benchmarks that represent the full state of the art in PL, FL and joint inference. We conduct experiments with four different use cases based on measurement data, and demonstrate how `Jewel` consistently outperforms all existing solutions for in-switch ML in terms of classification accuracy, while keeping resource usage under control. In particular, our fully joint PL-FL design substantially enhances the steadiness of `Jewel`, which performs well in all settings, whereas benchmarks have much more erratic results across use cases. By averaging over all scenarios, `Jewel` yields 3.2% higher accuracy than the second-best model proposed to date.

II. Related work

The proliferation of commercial programmable data planes has sparked a strong interest in the possibility of running ML models at line rate on individual packets. Due to the limited memory and computational capabilities of programmable network hardware, the attention has focused on the user-plane integration of models that are designed and trained offline [15].

Proposals differ in terms of the target network equipment and class of ML model. While several programmable user planes have been considered, *e.g.*, FPGA accelerators [34], NetFPGAs [17], or SmartNICs [35], prior works have primarily considered programmable switch ASICs based on the Protocol Independent Switch Architecture (PISA). The reason is that switches play a central role in any network infrastructure, and augmenting them with inference capabilities maximize the flexibility of any user-plane ML-driven operation. Programmable switches are thus also the focus of our work.

Due to the especially constrained environment that switch ASICs represent, and to the inherent properties of the PISA template, there is wide agreement that simple but effective DT and RF models [36] are the best option for in-switch ML integration, and all state-of-the-art solutions presented to date adopt such an approach. Table I presents a simple taxonomy of all main proposals for in-switch DT and RF models. Apart from a minority of solutions only supporting DTs and not more complex and accurate RFs [20], [22], [23], or that did not undergo the critical test of an implementation in hardware [22], [25], [26], [28], there exist three main gaps in the literature.

First, as anticipated in Section I, almost all models target PL or FL inference, but not both. As already discussed, PL models [16]–[23] are simpler and less resource-demanding, but prone to lower classification accuracy; instead, FL models [24]–[31] leverage flow features to achieve higher accuracy but occupy more memory in the switch and more importantly fail to observe short flows or early packets in long flows. To show the limitations of PL-only and FL-only models, we select three as representative of the state of the art in the performance evaluation of `Jewel` in §IV: Planter [18], which proposes an effective RF mapping strategy that has been adopted by several other works, and is also employed by `Jewel`; Mousika [20], a highly original solution based on a teacher-student ML approach leading to an extremely compact binarized DT implementation; and, Flowrest [31], the most complete model to date for in-switch FL inference. The only existing solution supporting concurrent PL and FL inference is NetBeacon [33], which adopts a strategy of deploying in the switch separate DT and RF models to perform the two types of classifications. As mentioned in Section I, we follow a different strategy in `Jewel`, and devise in §III a unified RF model that performs both PL and FL classification based on the nature of the packet under examination. In order to demonstrate the higher efficiency of this full joint PL+FL design, we include NetBeacon in our choice of benchmarks for comparative evaluation in §IV.

The second gap in the literature highlighted by Table I concerns the fact that very few solutions consider a hardware-aware design of the ML model. Only the work by Friday *et al.* [30] and Flowrest [31] attempt to include the constraints of the target programmable network equipment into the way the RF is planned before training. More precisely, Friday *et al.* apply Gray coding to the feature table codes, which allows them to reduce TCAM usage; Flowrest introduces instead rules to trim long time-dependent FL features. We build on these previous experiences, and propose a more comprehensive strategy to hardware-aware ML model design in §III.

The final gap we identify refers again to the model design stage, as most previous studies take the model preparation phase for granted, and rather focus on the user-plane part of the problem. Only a few works have made proposals for feature reduction, based on greedy algorithms informed by the feature importance returned by RF training [24], manual filtering [33], or collinearity analyses [29]. Yet, these are simple strategies that do not take into account hardware limitations, and are completely oblivious to model hyper-parametrization. To address this limitation, in §III we provide a systematic and automated pipeline for feature engineering and model hyper-parametrization that, consistently with the previous point, is mindful of the requirements imposed by the programmable switch. We will prove in §V that our hardware-aware feature and model preparation pipeline is paramount to identify models that grant high accuracy while fulfilling the constraints of the programmable switch –not only for `Jewel`, but also for the other ML models we compare against.

Ultimately, when compared to currently available approaches and as evidenced in the last line of Table I, `Jewel` combines joint PL and FL inference with a smart RF model design, in addition to being validated with production-grade programmable switches and released in open source. We thus believe that `Jewel` represents the most complete ML model for in-switch inference proposed to date, and the experimental evaluation in §V proves its consistently better performance over state-of-the-art PL, FL and PL+FL classifiers.

III. Joint in-switch flow-packet classification

We present the design and operation of our proposed `Jewel` by first providing a summary of the solution (§III-A), and then detailing the RF model for fully joint PL-FL inference (§III-B), the automated feature and model selection routines (§III-C), and the final implementation in hardware (§III-D).

A. Solution overview

Figure 1 recapitulates the operation of `Jewel` from a high-level perspective, which is fashioned after that of recent FL inference models for programmable switches [24], [31]. At step ①, the model is prepared and trained in the control plane via a software tool that explores all the possible combinations of hyper-parameters and features, takes into account the hardware constraints, optimizes the memory usage, and finally selects the model with the best F1 score. The model is then mapped during step ② into the PISA architecture via P4, built and injected into the pipeline by the controller.

The user plane implementation that performs the actual inference on incoming packets, in ③, is composed of three main elements. The traffic filtering, in step ④, identifies the target flows subject to inference (which may not represent the entirety of traffic) and extracts PL features from the relevant headers. The flow management, in ⑤, maintains a flow tracking register about the flows for which FL features are being collected, and stores and updates such FL features that are in preparation. Finally, the joint inference, in ⑥, exploits PL and FL (if available) features to classify the packet or flow.

The yellow arrows in Figure 1 represent five different paths that incoming packets can follow in the switch pipeline, according to their order of arrival in the flow they belong to. Assuming that FL inference is triggered at packet number `n` in the flow, all early packets from 1 to `n-1` are processed for PL feature extraction, and go through flow management where they are used to update the FL features that are under preparation for the flow. Then, these packets go through inference using only the available PL features. The packet number `n` in the flow follows the same path as the earlier ones, but then also retrieves the final value of the FL features after having updated them. Packet `n` thus undergoes inference

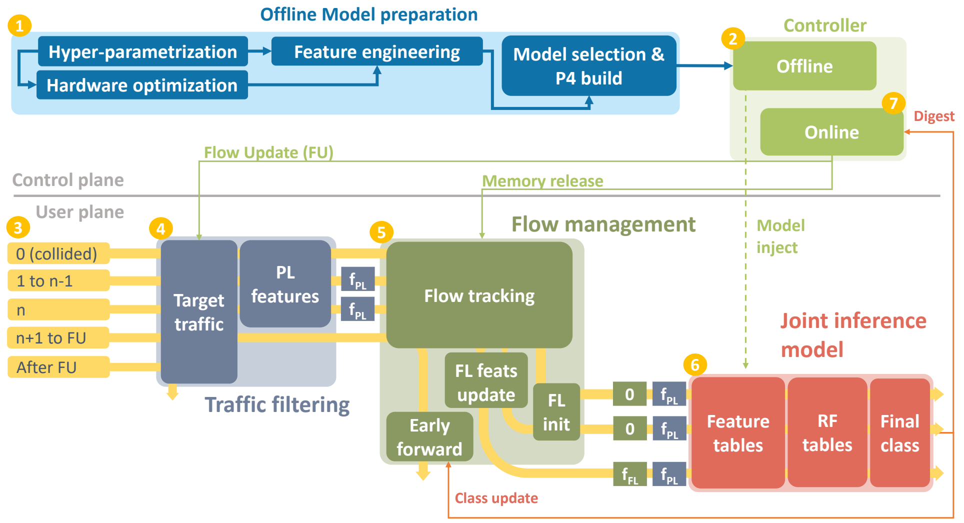


Figure 1: Overview of `Jewel` operation. Steps ①-⑦ denote the time sequence of events.

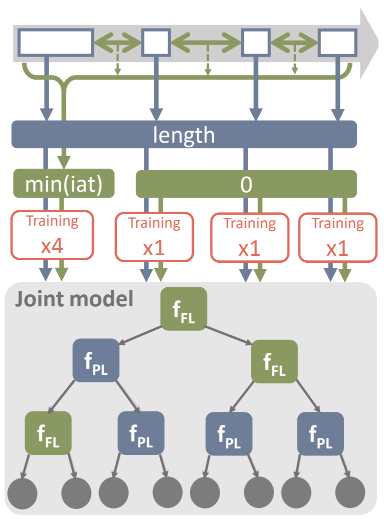


Figure 2: Joint PL-FL RF example.

with the full set of PL and FL features. The result of this classification is particularly important, as it determines the class that will be assigned to all packets that arrive after `n`. As seen in Figure 1, packets from `n+1` are redirected by flow tracking to an early forwarding table where they are tagged with the class determined for the whole flow based on `n`.

Upon classification, a digest is sent to the controller with the information of the class of the flow. The controller then triggers two actions, in ⑦: (*i*) a Flow Update (FU) that inserts directly into the target traffic table the egress port of the target flow, based on the class according to a predetermined policy (*e.g.*, malicious packets may be redirected to a honeynet for further analysis); and (*ii*) the release of the flow tracking registers occupied by the flow. Therefore all incoming packets of the flow arriving after the FU are forwarded according to the traffic table only, as also illustrated in the figure.

A particular case is that of packets tagged as `0` in Figure 1, which belong to flows that *collide* in the flow management step. As a matter of fact, the flow tracking register is typically implemented as a hash table with a finite size, meaning that two different flows may be mapped to the same hash key. In case of such a collision, the packet cannot initiate the FL operations, and `Jewel` simply proceeds to inference using PL features only. It is worth noting that, instead, pure FL solutions cannot classify all packets in collided flows.

B. Fully joint PL-FL model design

At the core of `Jewel` is an original ML model design that can perform both PL and FL inference via a single RF, which automatically adapts the mode of operation depending on the position of the packet within the flow. Thus, the component ⑥ in Figure 1 is realized with a single, monolithic RF.

To this end, we devise an RF model that receives as input a complete PL and FL feature set, and operates according to a simple but effective idea illustrated by means of a toy example in Figure 2, and described in detail next.

- During the training phase, for packets from `1` to `n-1` the feature set is composed of the actual PL features of the packet (*e.g.*, the packet length in the sample), plus *default* constant values never attained in reality for all FL features (*e.g.*, a null inter-arrival time in the example).
- When training gets to packet `n`, for which FL features become available, the feature set is switched to actual values of both PL and FL features (*e.g.*, in the example, the packet length but also the real minimum inter-arrival time measured on the first four packets of the flow).
- An important observation is that we assign *weights* to individual samples (*i.e.*, packets), so as to consider that their inference has not the same impact on the final result. Specifically, the first `n-1` packets are each given a weight 1, as their correct classification entails that one packet will be accurately tagged by the ML model during the actual inference phase. Instead, packet `n` is attributed a higher weight, as the way it is classified affects all following packets in the flow, which, as explained in §III-A, inherit the flow-level class. Specifically, we set the weight to `n` (*e.g.*, 4 in the toy example), which allows balancing evenly the attention of the model between its PL and FL inferences[1].

By adopting the strategy above, we teach the RF model to rely on PL features only when FL features are assigned the default values. Indeed, such default values are constant for all packets and completely uncorrelated to the variable to predict, such as the category or adversary nature of the packet. In other words, the RF learns to discard default values for FL features. Of course, when actual FL features are available, the RF also learns to use them to improve the classification accuracy. The operation during inference is then straightforward, and is depicted at the input of block ⑥ in Figure 1: whenever the

[1]We also experimented with other weight configurations, such as the total number of packets in the flow, yet those exceedingly downplayed the importance of PL inference, leading to reduced performance overall.

FL features are not available, *i.e.*, for packets 1 to n-1 or in collided flows, we use default FL feature values instead.

We refer to our approach as *fully joint*, as it uses an actual single model and in contrast with NetBeacon [33], which is the only solution for joint PL-FL inference in the literature and that relies on separate models for PL and FL classification. As we will show in §V, the fully joint design of Jewel allows for a more compact yet performing model that achieves substantial gains in both accuracy and resource usage over NetBeacon.

C. Hardware-tailored feature and model selection

A second key element of the design of Jewel is the original execution of the offline model preparation phase, marked as ① in Figure 1. We develop an automated model analysis tool that explores multiple combinations of (*i*) features and (*ii*) hyper-parameters like the maximum tree depth, maximum number of leaves, number of trees and value of n, *i.e.*, the rank of the packet for which the FL inference is triggered.

Our pipeline selects such variables while considering *by design* the inherent constraints of the underlying hardware, ensuring that the selected model can be ultimately deployed in the switch using the mapping methodology later detailed in §III-D. In particular, the hardware restrictions Jewel must abide by are depicted in the bottom-left part of Figure 3, and are linked to the limits in the bit size of three elements of the PISA architecture: (*i*) we use RegisterActions (RAs) to store the model features, hence the length of each feature value cannot exceed AR_LIM; (*ii*) as we evaluate tree split conditions using range matches in feature Match-Action Tables (MATs), the feature length cannot exceed RANGE_LIM; (*iii*) the TER_LIM limits the length of the codeword used to represent each path in a tree and matched via TCAM.

Ultimately, the feature and model selection tool employed by Jewel is implemented as a control-plane software, and is structured in the three phases detailed in Figure 3. First, we perform the model *hyper-parametrization* Ⓐ, by collecting all possible configurations hp(n,t,d) of the rank of the FL trigger packet n, the number of trees t in the RF, and the maximum depth of each tree d. These configurations are passed on to the hardware optimization Ⓑ, which, for each hp(n,t,d), assesses multiple options for the maximum number of leaves L in each tree. The value of L maps to the length of the codeword used in the TCAM, which, as anticipated, is bounded by TER_LIM (plus one); yet, smaller values of L are possible and actually recommended, as they save expensive and scarce TCAM resources. Importantly, the logical TCAM is implemented as multiple physical memories with a same size TER_SIZE, and such memories cannot be shared, implying that not all values of L are sensible. Namely, it only makes sense to test values of L with increments equal to TER_SIZE, since smaller increments do not save any TCAM but may unnecessarily restrain the model size and curb its performance. Then, the function Range returns the set of values of L that align with such increments.

For every combination of hp(n,t,d) and L, a *seminal model* RF_a is first trained with all available features at both packet- and flow-level, *i.e.*, (PL,FL). Note that the weight assigned to packet n during training (as explained in §III-B) is considered as a system parameter w in the model selection process, for the sake of flexibility; as already mentioned, we found in fact that w=n worked well across all our tests. The trained seminal model RF_a is used in the feature engineering Ⓒ phase to rank features based on their Mean Decrease in Impurity (MDI) importance for the classification task. This allows generating feature sets of increasing size (from 1 to the total number of features), by including the ranked features one by one starting with the most important one. Models RF_b are then trained by sweeping through all feature sets, and saved along with their performance (*e.g.*, F1 scores).

The complete nested cycles through all n, t, d, L and feature sets executes an exhaustive search over the RF models design space. All trained models produced are ranked based their accuracy, and the best one selected. If the best RF model uses FL features that involve time (as it is often the case), one final step is enacted, also as part of the feature engineering block Ⓒ in Figure 3 and to ensure hardware consistency. Specifically, time-based features are calculated in the switch via timestamps with a length higher than AR_LIM and RANGE_LIM, with RANGE_LIM<AR_LIM. Hence, we adopt a strategy for feature representation proposed by Flowrest [31], and trim down the feature length by removing the most significant bits such that the feature length is reduced to AR_LIM; we then further reduce the length by removing the least significant bits to limit the feature length to RANGE_LIM. In fact, we also explore possibilities of further optimization of the length of time features, by training the selected model with all feasible trimmings within RANGE_LIM, as well as all possible combinations of trimmed sets in presence of multiple time features. This returns the best set possible of hardware-compliant ML model.

Finally, we run a Select function on the set of returned models, which picks the single best solution for implementation in the switch, based on a simple performance score

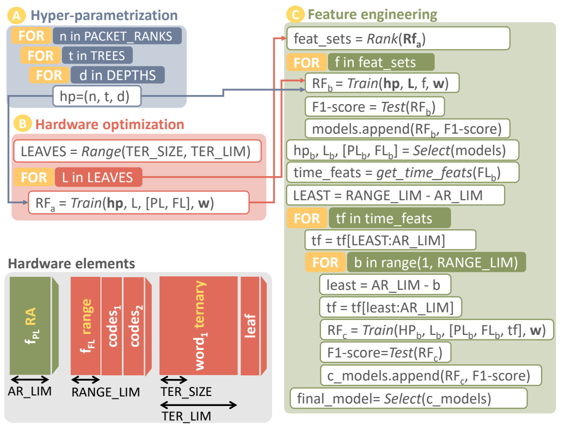
$$\alpha\frac{1}{3}\left(\mathrm{F1}_{\text{micro}}+\mathrm{F1}_{\text{macro}}+\mathrm{F1}_{\text{weighted}}\right)+(1-\alpha)(1-\rho), \quad (1)$$

where $\mathrm{F1}_{\text{micro}}$, $\mathrm{F1}_{\text{macro}}$ and $\mathrm{F1}_{\text{weighted}}$ are three metrics of inference accuracy based on the F1 score that can be derived on the training data as detailed in §IV-D, ρ is a measure of memory footprint of the model, expressed as a fraction of the total available resources in the target switch, and α is a parameter used to weight the two contributions of accuracy and resource usage. In our experiments, we employ the fraction of consumed TCAM for ρ, since that is arguably the most scarce and expensive resource in our target hardware, and we set α to 0.5 so as to balance fairly the two components.

D. In-switch implementation and operation

Figure 4 shows the mapping of Jewel onto a PISA pipeline. For simplicity, the figure depicts a toy example of a model with two features, one PL and one FL, and an RF of two trees.

The top layer in the figure depicts the sequence of MATs and RAs traversed by the packets in the different stages. The



```
A Hyper-parametrization
FOR n in PACKET_RANKS
  FOR t in TREES
    FOR d in DEPTHS
      hp=(n, t, d)
B Hardware optimization
LEAVES = Range(TER_SIZE, TER_LIM)
FOR L in LEAVES
  RF_a = Train(hp, L, [PL, FL], w)
C Feature engineering
feat_sets = Rank(RF_a)
FOR f in feat_sets
  RF_b = Train(hp, L, f, w)
  F1-score = Test(RF_b)
  models.append(RF_b, F1-score)
hp_b, L_b, [PL_b, FL_b] = Select(models)
time_feats = get_time_feats(FL_b)
LEAST = RANGE_LIM - AR_LIM
FOR tf in time_feats
  tf = tf[LEAST:AR_LIM]
  FOR b in range(1, RANGE_LIM)
    least = AR_LIM - b
    tf = tf[least:AR_LIM]
    RF_c = Train(HP_b, L_b, [PL_b, FL_b, tf], w)
    F1-score=Test(RF_c)
    c_models.append(RF_c, F1-score)
final_model= Select(c_models)
```

Figure 3: Hardware-aware model selection process.

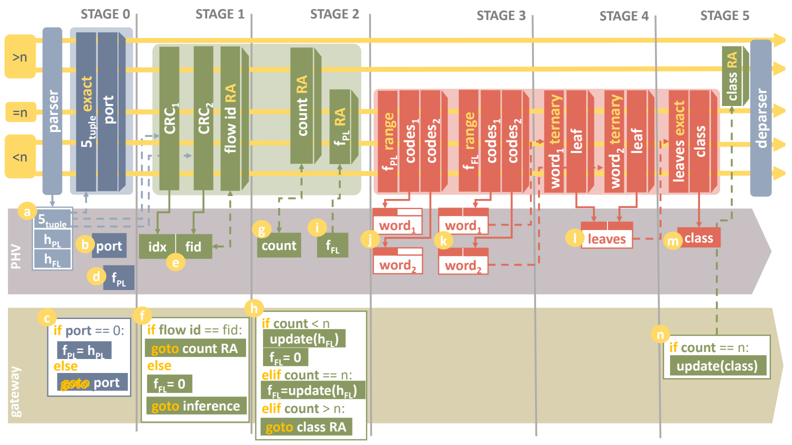


Figure 4: Detail of the mapping of `Jewel` into the PISA archictecture.

packets entering the switch are divided into three groups, highlighted in yellow. From bottom to top, we see the packets before `n`, packet `n`, and those after `n`. From the groups, five arrows depart and traverse the pipeline from left to right. Such arrows identify the different paths that can be followed, consistently with the operation discussed in Figure 1.

In the middle layer, the Packet Header Vector (PHV) carries the headers extracted from the packets and the metadata variables generated at different points of the pipeline. The interactions between the top layer and the PHV are also portrayed as arrows. Dashed arrows refer to matching, in the case of MATs, and comparison or updates in the case of RAs. Solid arrows indicate an assignment to metadata.

The bottom layer shows the Gateway where the if/else conditions applied to the data of the PHV reside. The interactions between the PHV and the Gateway is highlighted with encircled letters, from ⓐ to ⓜ.

When a packet enters the switch pipeline, three relevant header fields are extracted: the 5-tuple characterizing a flow, composed of source and destination IP addresses, source and destination port and protocol; the header field $\mathtt{h}_{\mathtt{PL}}$ used as the PL feature; the header field $\mathtt{h}_{\mathtt{FL}}$ employed to calculate the FL feature. The 5-tuple is matched against the flow table to understand whether the flow a target for inference, and, upon a match, the egress port is retrieved. A value of the port different from zero means that the flow has already been classified and the packet is forwarded with no further action. Otherwise, the value of $\mathtt{h}_{\mathtt{PL}}$ is assigned to the PL metadata $\mathtt{f}_{\mathtt{PL}}$.

In the following stage, the 5-tuple is used to compute two hashes, of 16 and 32 bits, to calculate the flow registers index `idx` and the flow identifier `fid`, respectively. The index is used to access the entries of the RAs allocated to the flow. Three RAs store respectively, the flow id, the count of the packets, and the FL features $\mathtt{f}_{\mathtt{FL}}$ updated to the last packet seen. If the `fid` is the same as the flow identifier in the RA (or if the latter is empty) the packet is sent to the count RA. Otherwise, a flow collision is detected, $\mathtt{f}_{\mathtt{FL}}$ is set to `0` and the packet is advanced to the inference stage for PL classification.

In the next stage, the count RA is incremented and three different cases are identified according to its updated value. If it is less than `n`, the $\mathtt{f}_{\mathtt{FL}}$ RA is updated using $\mathtt{h}_{\mathtt{FL}}$ and $\mathtt{f}_{\mathtt{FL}}$ is set to 0, *i.e.*, we mark the packet for PL inference. If the count is `n`, the $\mathtt{f}_{\mathtt{FL}}$ RA is updated and its value assigned to $\mathtt{f}_{\mathtt{FL}}$ metadata, *i.e.*, we prepare the packet for FL inference. If the count is greater than `n`, the packet is sent to the class RA, where it is forwarded based on its already determined class.

Starting from the next stage, the joint model is encoded in a sequence of MATs. We adopt a mapping of the RF to the PISA pipeline that is mimicking that first proposed by Planter [19]. In the toy example in Figure 4, one MAT per feature and one MAT per tree are employed, such that two feature MATs and two tree MATs are represented. Each feature MAT encodes the split decisions (*i.e.*, conditions on whether one feature is higher or lower than a trained threshold) of both trees. The splits are encoded in the form of ranges, with the first range covering values from 0 to the smaller threshold present in the splits of the target feature, and each subsequent range encompassing the values up to the following threshold. Each range match can then be associated to an action, *i.e.*, a code, for each tree, which encodes the outcomes of all the splits where the feature occurs. For instance, in Figure 4, when $\mathtt{f}_{\mathtt{PL}}$ is matched against the PL feature MAT, $\mathtt{code}_1$ and $\mathtt{code}_2$ are retrieved to partially fill $\mathtt{word}_1$ and $\mathtt{word}_2$, respectively. The same process is repeated with $\mathtt{f}_{\mathtt{FL}}$ against the FL feature MAT, filling completely $\mathtt{word}_1$ and $\mathtt{word}_2$. The metadata variables $\mathtt{word}_1$ and $\mathtt{word}_2$ encode the outcomes of all the splits of the first and the second tree, respectively.

In the next stage, the metadata variables $\mathtt{word}_1$ and $\mathtt{word}_2$ are matched against their respective tree MATs. Each tree MAT encodes all the different combinations of split outcomes leading to a different leaf of the tree. Such combinations are masked using ternary matches according to the specific subset of bits, *i.e.*, splits, that lead to a specific leaf. In Figure 4, $\mathtt{word}_1$ and $\mathtt{word}_2$ are matched against their respective tree MATs and fill a `leaves` metadata that contains the classes predicted by each of the two trees.

In the final stage, the `leaves` metadata is exactly matched against a final table encoding all combinations of possible predictions by the two trees. The class associated with each match is the most voted one. All the cases with no majority are instead encoded as MAT entries that assign as the final class that of the tree with the highest accuracy.

IV. Experimental setup

We implement `Jewel` in an experimental platform with industry-grade programmable switches (§IV-A) along with four state-of-the-art benchmarks (§IV-B). We then run comprehensive tests with four tasks based on real-world traffic (§IV-C), collecting relevant performance metrics (§IV-D).

A. Testbed setup

We run all experiments in a network testbed equipped with 3 switches and 2 servers. The switches are Edgecore Wedge 100BF-QS programmable switches, with Intel Tofino BFN-T10-032Q chipsets and 32 100GbE QSFP28 ports. They all run the Open Network Linux (ONL) operating system, and the Intel Software Development Environment (SDE) version 9.7.0 used for compiling P4 programs. The SDE also comes with the Intel P4 Insight tool [37] which provides an interactive graphical user interface with a detailed analysis of the mapping of P4 programs onto specific switch resources. We employ the tool to analyze all P4 programs in terms of the resource usage.

The two servers feature Intel 8-core Xeon processors running at 2GHz, with 48GB of RAM, and QSFP28 interfaces, hence the testbed is a full 100-Gbps platform. We use one server to implement the control plane. Specifically, it hosts a controller that binds to the Barefoot Runtime Interface (BRI) of the switch and performs the initial setup of the switch, activating ports and loading the trained RF models as table entries via a P4 program. The controller also serves as a collector for classification performances from the experiments. The second server runs traffic sources and sinks. On the one hand, it injects measurement data from the classification use cases presented next, by replaying `pcap` traces using Tcpreplay [38]. On the other hand, it employs the MoonGen packet generator [39] to inject background traffic at 100 Gbps, simultaneously to the use case traffic. This background traffic is not a target for inference, hence it is advanced by the traffic filtering module in Figure 1 to the regular forwarding table of the switch. The rationale for having it is to demonstrate that the tested models can achieve line-rate inference even in presence of substantial concurrent non-target traffic.

B. Benchmarks

We compare `Jewel` against four recent solutions for in-switch ML that are representative of the state of the art across DT and RF models, as well as PL and FL inference strategies. The rationale for our choice of benchmarks is outlined in §II, and leads to two PL models, *i.e.*, Planter [18] and Mousika [20], one FL model, *i.e.*, Flowrest [31], and the only PL+FL model available in the literature, *i.e.*, NetBeacon [33].

It is important to highlight that none of these models include a systematic model selection routine like that employed by `Jewel`. As we will later show in the experimental evaluation, the identification of the best model is paramount to ensure the optimal operation of the ML model. Therefore, for the sake of fairness, we adapt the model selection procedure described in §III-C so that it can inform the choice of (i) the best PL-only model, which is then employed by both Planter and Mousika, and (ii) the best FL-only model, used by Flowrest. As far as NetBeacon is concerned, we use both (iii) the best PL model with one tree to implement the DT that NetBeacon adopts to classify individual packets, and (iv) the best FL model with one tree to implement its per-flow DT classifier. In all these cases, we explore the hyper-parameters in the model selection blocks Ⓐ and Ⓒ of Figure 3, whereas we consider the check on `L` as an original contribution and unique feature of `Jewel`.

Overall, our aim is to put the benchmarks in the best position possible to compete with `Jewel`. Yet, that also means that the performance we report in §V for Planter, Mousika, Flowrest, and NetBeacon represents best scenarios that would be hard to identify without the help of our novel model selection routine. We next detail the benchmark implementations.

Planter [18]. As the source code of Planter is not publicly available at the time of writing, we replicate the proposed PL solution using the code made available by Henna [40], which is an implementation of the same strategy by different authors.

Mousika [20]. We implement Mousika using the publicly available artifacts [41]. Following the strategy proposed by the original model, we take the best PL configuration identified by our automated model selection routine, binarize the selected PL features, and then use them to train the teacher RF model. The teacher is finally distilled into student model in the form of a binary decision tree. Given the very compact representation of the binary tree, we enforce no limits on its size.

Flowrest [31]. We use the public source code [42] to encode the models for FL inference in the switch.

Netbeacon [33]. Also here, we rely on publicly available source code [43] for the implementation. It is worth noting that although the paper describes how the mapping scheme proposed can be used to map RF models with multiple trees, this capability is unclear from the model design, nor is it demonstrated, as the artifacts released [43] only employ single-tree RFs –even for a use case where the RF was indicated to have more trees. Since it is also not obvious to us how the model can be extended to operate with multiple trees without exceeding the memory limits of the switch, we did not have a choice but to restrain our feature and model selection tool to use RFs with 1 tree for this benchmark.

C. Use cases

We select a range of measurement-based classification tasks, which build on open datasets and allow assessing the robustness of the different solutions across heterogenous scenarios.

UNIBS [44], [45] is a service classification dataset containing web (HTTP and HTTPS), mail (POP3, IMAP4, and SMTP), peer-to-peer applications (BitTorrent and Edonkey), and other protocols (FTP, SSH, and MSN). The traces, consisting of traffic generated by 20 workstations, are collected at the edge router of the University of Brescia's campus network, covering three consecutive working days. To classify 8 different traffic services, we train the models with the first day of traffic and test it with the second day.

UNSW-IoT [46] revolves around device identification, and comprises measurement data that encompasses traffic flows generated by a wide variety of devices, including a diverse range of both IoT and non-IoT devices. The captures are collected within a living lab emulating a smart environment. In order to identify 26 device types, we use 15 days of data for model training and one day of data for model testing.

IoT-23 [47] is a malicious traffic detection use case, and aims to provide researchers with an extensive dataset comprising real and labeled IoT malware infections, as well as IoT benign traffic, to develop machine learning algorithms. The dataset contains 20 malicious and 3 benign captures spanning 2018 and 2019. Such malicious and benign traffic is captured from infected IoT devices and regular real IoT devices, *i.e.*, a Philips HUE smart LED lamp, an Amazon Echo home intelligent personal assistant, and a Somfy smart doorlock, respectively, in a controlled environment. We use the IoT-23 dataset to detect 4 benign and 10 malware traffic classes.

ToN-IoT [48] captures benign traffic and 9 types of cyber-attacks, collected from a representative medium-scale testbed deployed using several virtual machines such as hosts of Windows, Linux, and Kali Linux operating systems to manage the interconnections between the three Cloud, Fog, and Edge network domains. The measurements were carried out with seven IoT and industrial IoT (IIoT) devices, *i.e.*, weather and Modbus sensors, a thermostat, a motion light, a garage door, a GPS tracker, and a fridge, and also some non-IoT devices such as two iPhone 7 and a Smart TV.

D. Metrics

We evaluate the classification accuracy of `Jewel` and of the benchmarks using the widely adopted F1 score (F1) metric. This metric can be computed out of three principal measures of a classification problem, *i.e.*, true positives (TP), false positives (FP), and false negatives (FN), as F1=2TP/(2TP+FP+FN). We average F1 in three different ways: (*i*) a *micro* average that is computed by summing up the TP, FP, and FN from all classes and then calculating the metric; (*ii*) a *macro* average that is the mean of individual class scores; and, (*iii*) a *weighted* average which weights individual class scores by the number of samples present in the dataset. Each average has a different meaning: micro, macro and weighted values quantify how accurate a model is in classifying individual packets, whole classes, and classes weighted by their support, respectively.

To ensure a fair comparison between the benchmarks, we compute TP, FP, and FN on a per-packet basis so that all solutions are judged on the same number of samples. This is straightforward in the PL models, where packets are the actual unit of operation. In the case of the FL or joint models, we apply a per-packet basis by extending the result of the classification of a flow to all packets ensuring the one that triggered the FL inference classification point (*e.g.*, after packet `n` in `Jewel`). Also, all early flow packets go unclassified by FL models are considered not to be correctly classified, as the inference is completely oblivious to them.

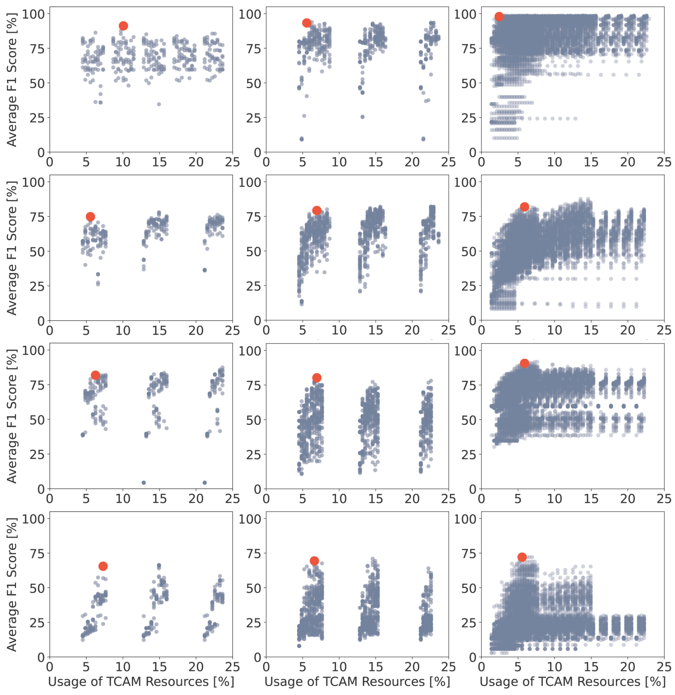


Figure 5: Scatterplots of the average micro, macro and weighted F1 versus TCAM usage, for all models searched by our proposed automated routine. Red dots highlight the selected model for all combinations of UNIBS, UNSW-IoT, IoT-23 and ToN-IoT use cases (top to bottom) and PL, FL, and PL+FL (columns) approaches. The density grows from left to right as the search space is enlarged to include additional hyper-parameters `n` in FL models and `L` in PL+FL models.

V. Experimental Results

We start our evaluation by demonstrating the critical role of the systematic and automated feature and model selection scheme that we propose in §III-C. Figure 5 shows the average of the three F1 (on the y axis) and TCAM resource usage (on the x axis) computed for every single model (dot) explored by our routine. The final model is selected as the one that best balances the two metrics, as per (1). The figure shows each combination of benchmark (columns) and use case (rows), and highlights the picked model as a larger red dot.

These plots show that the space of possibilities for hyper-parametrization is extremely large, and a manual selection of the best-performing solution across the hundreds or thousands of options severely risks to be sub-optimal. Instead, a structured and automated process can pinpoint the RF model that provides the best trade-off of accuracy and resource footprint, denoted by the red dots that are then implemented in hardware.

The tests with real-world programmable switches highlight the superiority of `Jewel` across all scenarios in terms of classification accuracy. Table II juxtaposes the different models in terms of the average of their micro, macro and weighted F1.

	Mousika	Planter	Flowrest	NetBeacon	Jewel
UNIBS	90.351%	91.560%	96.398%	94.570%	**98.354%**
UNSW	82.003%	79.853%	80.691%	78.594%	**87.317%**
ToN-IoT	27.554%	70.496%	73.461%	70.063%	**75.703%**
IoT23	86.054%	88.147%	82.857%	86.076%	**91.314%**

Table II: Average of `F1` metrics across models and use cases.

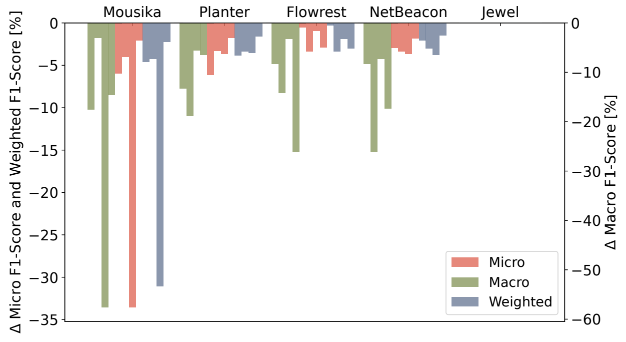


Figure 6: Accuracy loss of the benchmarks across `F1` metrics.

Our solution is invariably the best, with gains in the range 2.0–5.3% with respect to the second-ranked model, and typically of 10% or more over the worst benchmark. These are not negligible improvements, considering that the competitors are all of the best solutions currently available, and that `Jewel` outperforms them across very heterogeneous use cases.

In fact, *consistency* in good performance is a remarkable characteristic of our model. Indeed, the second-best model mentioned above changes for each use case: Flowrest yields better accuracy than NetBeacon, Planter and Mousika in the tasks proposed in the UNIBS and ToN-IoT datasets, but Mousika and Planter take that role across the other two use scenarios, with second-best accuracy in UNSW and IoT-23, respectively. Each state-of-the-art model has thus properties that allow it to surpass other benchmarks in a specific scenario, yet penalize it in others. This is not the case for `Jewel`.

Comparing `Jewel` directly to NetBeacon *i.e.*, the only other PL+FL solution, `Jewel` always performs better, with gains in the range 3.8–8.7%. We attribute this to two reasons. First, as NetBeacon's FL models are all single-tree RFs, their accuracy is limited compared to `Jewel`'s multiple-tree RF. Second, in NetBeacon, the PL model is used each time FL inference cannot be performed. Thus, many packets might only be classified at PL, resulting in an overall lower accuracy compared to `Jewel` that instead prioritizes FL inference.

We have a deeper look at the unwavering quality of our model across diverse settings in Figure 6, which breaks down the result above across micro, macro and weighted `F1`. To better appreciate the advantage of `Jewel`, the plot shows the difference in performance between each model and the model that performs the best. It is apparent how our proposed solution is steadily achieving not only the highest average classification accuracy across all `F1` metrics, but also for each and every individual metric. In other words, `Jewel` invariably leads a range of state-of-the-art in-switch inference models when considering the quality of classification at both the level of individual packets, or at that of whole flows. The result is impressive, considering that optimizing different `F1` is a hard task: this is evidenced in Figure 6 by the fact that all benchmark exhibit substantial loss of accuracy for one `F1` metric or another, in one or multiple use cases.

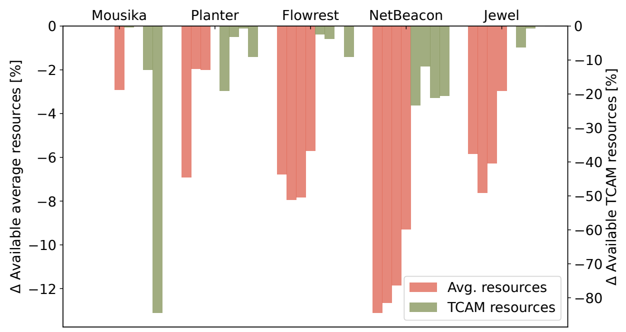


Figure 7: Reduction of available total and TCAM resources with respect to the most parsimonious model in each use case.

Finally, we look at resource usage, which is also an important parameter when considering resource-constrained user plane equipment, and it is indeed a parameter we consider in our model selection routine. Figure 7 shows the reduction of available resources with respect to the most frugal model, in terms of both total switch memory and expensive TCAM. As expected, PL models have often (but, again, not always) the smaller resource footprint, especially for the total memory. Yet, `Jewel` (*i*) is on par with an FL-only model like Flowrest, despite the added PL functionalities, and (*ii*) is much more resource-efficient than its direct PL+FL competitor, *i.e.*, NetBeacon. In fact, when looking at TCAM, *i.e.*, the target of our optimization during model selection, `Jewel` is often the most effective model, saving resources even over PL models.

Overall, the comparative evaluation highlights how `Jewel` offers an excellent trade-off between the best accuracy and a resource-efficient implementation, largely outperforming its direct competitor, *i.e.*, NetBeacon. This lets us claim that `Jewel` sets a new state of the art for in-switch ML inference.

VI. Conclusions

We proposed `Jewel`, an original model for joint packet- and flow-level in-switch inference, based on a single resource-efficient random forest trained to address both types of classification and a first-of-its-kind systematic and automated model hyper-parametrization scheme. Extensive experiments prove the high accuracy and strong consistency of `Jewel` across a range of use cases, establishing it as a new standard for in-switch ML. The authors have provided public access to their code at https://github.com/nds-group/Jewel.

Acknowledgments

This work has received funding from the CHIST-ERA grant no. CHIST-ERA-20-SICT-001 "ECOMOME", via grant PCI2022-133013 of Agencia Estatal de Investigación, from the Spanish Ministry of Science and Innovation through grant no. PID2021-128250NB-I00 ("bRAIN"), and from the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union-NextGenerationEU through the UNICO 5G I+D project no.TSI-063000-2021-52 "AEON-ZERO".

References

[1] E. Uhlemann, "Time for autonomous vehicles to connect [connected vehicles]," *IEEE Vehicular Technology Magazine*, vol. 13, no. 3, pp. 10–13, 2018.

[2] A. Seam and A. Poll, "Enabling mobile augmented and virtual reality with 5g networks," *AT&T Foundry*, Feb 2017.

[3] J. Xu, K. Papangelis, J. Dunham, J. Goncalves, N. J. LaLone, A. Chamberlain, I. Lykourentzou, F. L. Vinella, and D. I. Schwartz, "Metaverse: The vision for the future," in *CHI EA '22*. NY, USA: ACM, 2022.

[4] A. A. Gebremariam, M. Usman, and M. Qaraqe, "Applications of artificial intelligence and machine learning in the area of SDN and NFV: A survey," *SSD 2019*, pp. 545–549, 2019.

[5] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2019.

[6] Y. Zhao, Y. Li, X. Zhang, G. Geng, W. Zhang, and Y. Sun, "A survey of networking applications applying the software defined networking concept based on machine learning," *IEEE Access*, vol. 7, pp. 95 397–95 417, 2019.

[7] P. Kalmbach, J. Zerwas, P. Babarczi, A. Blenk, W. Kellerer, and S. Schmid, "Empowering self-driving networks," in *SelfDN 2018*. NY, USA: ACM, 2018, p. 8–14.

[8] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 3, 2019.

[9] European Telecommunications Standards Institute (ETSI), "Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Report on enabling autonomous management in NFV-MANO," ETSI GR NFV-IFA 041, 2021.

[10] 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," ser. SOSR '15. NY, USA: ACM, 2015.

[11] Intel, "Tofino Programmable Ethernet Switch ASIC," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[12] Netronome, "Netronome Agilio SmartNICs," 2016. [Online]. Available: https://www.netronome.com/products/smartnic/overview/

[13] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014.

[14] D. R. K. Ports and J. Nelson, "When should the network be the computer?" ser. HotOS '19. NY, USA: ACM, 2019, p. 209–215.

[15] R. Parizotto, B. L. Coelho, D. C. Nunes, I. Haque, and A. Schaeffer-Filho, "Offloading machine learning to programmable data planes: A systematic survey," *ACM Comput. Surv.*, jun 2023, just Accepted.

[16] 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*, 2022.

[17] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *HotNets 2019*. ACM, 2019.

[18] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *SIGCOMM '21*. NY, USA: ACM, 2021, p. 12–14.

[19] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Automating in-network machine learning," *arXiv*, 2022.

[20] 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*, pp. 1938–1947.

[21] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Henna: Hierarchical machine learning inference in programmable switches," in *NativeNI 22*. ACM, 2022, p. 1–7.

[22] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards network-accelerated ML-based distributed computer vision systems," in *IEEE ICPADS*, 2021, pp. 122–129.

[23] 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 *SRDS*, 2022.

[24] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-network inference with random forests," *CoRR*, vol. abs/1909.05680, 2019.

[25] J. Lee and K. P. Singh, "Switchtree: in-network computing and traffic analyses with random forests," *Neural Computing and Applications*, pp. 1–12, 2020.

[26] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *IEEE INFOCOM 2021*, pp. 1–10.

[27] 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*, vol. 18, no. 4, pp. 4353–4364, 2021.

[28] B. Coelho and A. Schaeffer-Filho, "BACKORDERS: Using random forests to detect DDoS attacks in programmable data planes," in *EuroP4 '22*. NY, USA: ACM, 2022, p. 1–7.

[29] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, "INC: In-network classification of botnet propagation at line rate," in *Computer Security – ESORICS 2022*. Springer International Publishing, 2022, pp. 551–569.

[30] K. Friday, E. Bou-Harb, and J. Crichigno, "A learning methodology for line-rate ransomware mitigation with p4 switches," in *Network and System Security*. Springer Nature Switzerland, 2022, pp. 120–139.

[31] A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *IEEE INFOCOM 2023*.

[32] B. Hullar, S. Laki, and A. Gyorgy, "Early identification of peer-to-peer traffic," in *IEEE ICC*, 2011.

[33] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *32nd USENIX symposium on security*, 2023.

[34] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ml," *ASPLOS*, 2022.

[35] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, H. Haddadi, G. Antichi, and R. Bifulco, "Re-architecting traffic analysis with neural network interface cards," in *NSDI*. Renton, WA: USENIX, Apr. 2022.

[36] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" in *NeurIPS 2022*, New Orleans, United States, Nov. 2022.

[37] Intel, "P4 Insight," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.

[38] A. Turner and F. Klassen, "Tcpreplay," 2013.

[39] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," ser. IMC '15. NY, USA: ACM, 2015, p. 275–287.

[40] A.T-J. Akem et al., "Henna," https://github.com/nds-group/Henna.

[41] G. Xie et al., "Mousika," https://github.com/xgr19/Mousika.

[42] A.T-J. Akem et al., "Flowrest," https://github.com/nds-group/Flowrest.

[43] G. Zhou et al., "Netbeacon," https://github.com/IDP-code/NetBeacon.

[44] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli, "Detection of encrypted tunnels across network boundaries," *2008 IEEE ICC*, pp. 1738–1744, 2008.

[45] University of Brescia, "UNIBS internet traces," http://netweb.ing.unibs.it/~ntw/tools/traces/, 2009.

[46] 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, 2019.

[47] S. Garcia, A. Parmisano, and M. J. Erquiaga, "IoT-23: A labeled dataset with malicious and benign IoT network traffic," Data set, 2020. [Online]. Available: http://doi.org/10.5281/zenodo.4743746

[48] 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," 2020.