This is the author's accepted version of the journal article. The final version published is A. T.-J. Akem, B. Bütün, M. Gucciardo and M. Fiore, "Practical and General-Purpose Flow-Level Inference with Random Forests in Programmable Switches," IEEE/ACM Transactions on Networking, doi: 10.1109/TON.2025.3564465.

Practical and General-Purpose Flow-Level Inference with Random Forests in Programmable Switches

Aristide Tanyi-Jong Akem, *Member, IEEE,* Beyza Bütün, *Student Member, IEEE,* Michele Gucciardo, *Member, IEEE,* and Marco Fiore, *Senior Member, IEEE*

***Abstract*—Integrating machine learning (ML) models directly in the network user plane enables inference on data traffic at line rate, and can dramatically reduce the latency and improve the scalability of key functionalities like traffic classification or intrusion detection. Yet, the hardware that can be used to this purpose, in particular programmable switches, present stringent constraints in terms of limited memory and little support for mathematical operations or data types that render ML model deployment a substantial technical challenge. In this paper, we make a step forward in user-plane ML by introducing `Flowrest`, a solution that redefines the state of the art in flow-level inference for programmable switches. `Flowrest` allows implementing general-purpose Random Forest (RF) models in industry-grade switches by (i) suitably handling stateful flow-level (FL) features in the switch ASIC, (ii) achieving low-collision flow management, and (iii) customizing RF models right from the design phase for in-switch operation. We develop `Flowrest` as an open-source software using the P4 language and evaluate its performance in an experimental testbed with Intel Tofino switches. Experiments with inference tasks of varying complexity prove that our solution improves accuracy by over 10 percent points on average with respect to the second-best competitor out to five recent approaches for RF-based in-switch inference, while maintaining sub-microsecond latency.**

***Index Terms*—Programmable switches, machine learning, in-switch inference, network intelligence, random forest, P4.**

I. Introduction

NETWORK architectures are steadily growing in complexity to support modern digital applications, and are envisioned to become self-driving [1] so as to realize zero-touch network and service management (ZSM) [2], by having network controllers and orchestrators collect and analyze measurement data to design policies and algorithms that can make effective real-time or proactive management decisions. Ultimately, these paradigms reduce or even completely replace traditional human-in-the-loop management methods.

This research was supported by the SNS JU and the European Union's Horizon Europe research and innovation program under Grant Agreement No. 101139270 (ORIGAMI), by the Spanish Ministry of Science and Innovation through grant no. PID2021-128250NB-I00 (bRAIN), and by project PCI2022-133013 (ECOMOME), funded by MICIU/AEI/10.13039/501100011033 and the European Union "NextGenerationEU"/PRTR. M. Fiore is a Comunidad de Madrid Talent Attraction fellow (2023-5A/TIC-28944) and B. Butun is a Comunidad de Madrid predoctoral fellow (PIPF-2022/COM-24867). *(Corresponding author: Aristide Akem.)*

A. Akem is with the Department of Engineering Science, University of Oxford, OX1 3PJ, Oxford, UK, and M. Gucciardo is with NEC Laboratories Europe, 28108 Madrid, Spain. This work was done while they were both with IMDEA Networks Institute, 28918 Madrid, Spain. (email: aristide.akem@eng.ox.ac.uk, michele.gucciardo@neclab.eu)

B. Bütün and Marco Fiore are with IMDEA Networks Institute, 28918 Madrid, Spain. B. Bütün is also with Universidad Carlos III de Madrid, 28911 Madrid, Spain. (email: {beyza.butun, marco.fiore}@imdea.org)

On this journey, machine learning (ML) models can play an important role towards the automation of many management functions. Abiding by the principles of Software Defined Networking (SDN), ML models have been deployed in the network control planes, where they support, among others, traffic classification, quality of service (QoS) prediction, quality of experience (QoE) estimation, or traffic engineering and security, as thoroughly reviewed in several recent surveys [3]–[6]. While control-plane ML integrates intelligence in the network, it often requires communication to and from the user plane. This introduces structural overheads in the data exchange and delays in the decision-making process that prevent the ML models from running at line rate, *i.e.*, on the entirety of the transiting traffic and with no perceptible added latency. For instance, communication between network planes yields delays in the order of tens of milliseconds [7] whose bounding to tight guarantees is cumbersome and consumes hardware resources [8]; similarly, mirroring all packets to the control plane for in-depth analysis hardly scales to speeds of tens of Gbps. Due to these inherent limitations, control-plane ML fails to meet key requirements for self-driving networking [1], and may not fulfil next-generation mobile networks sub-millisecond end-to-end latency needs [9].

Line-rate inference requires pure user-plane implementations of ML models that are aligned with in-network computation principles [10]. The proliferation of commercial protocol-independent and programmable user planes like Intel Tofino ASICs [11] and Network Processing Units (NPUs) [12], alongside dedicated programming languages such as P4 [13], has led to a flurry of proposals to embed different ML models directly into switches or Smart Network Interface Cards (SmartNICs), as extensively discussed in a recent survey [14]. The endeavour is particularly challenging in the case of programmable switches, due to their severe constraints in terms of low available memory, limited support for mathematical operations and maximum allowable operations per packet [10], [15].

Early works have demonstrated the potential of in-switch inference, using Decision Tree (DT) and Random Forest (RF) models. However, as discussed in Section II, they have focused on packet-level (PL) classification, exploiting features easily extracted from packet headers. While PL operations are substantially easier to code in programmable user planes, inference at flow-level (FL) is much more relevant in a variety of networking tasks. In general, FL classification provides a more contextual understanding of network traffic [16], [17], as it can leverage interesting insights about the relationships between the packets of the same flow. Similarly, statistical FL features are more robust in the presence of increasingly

encrypted network traffic [18], and are less easily impacted by network security solutions [19]. Also, network-wide policies are generally applied on a flow or service basis, and not at packet level: this is the case, for instance, of traffic engineering [20], or QoS and QoE management [21], [22]. Despite its advantages, FL inference is involved to integrate in switching architectures that are designed to process one packet at a time.

In this paper, we present a comprehensive design that sets the current state of the art for FL inference in programmable switches. Our solution, named `Flowrest`, enables integrating large RF models in production-grade programmable hardware ASICs, so as to perform challenging inference tasks on individual traffic flows at line rate. The development of `Flowrest` entails the following major contributions.

- We present a practical framework for FL inference that is tailored to programmable switch architectures. Our framework encompasses both the off-line model design and on-line model operation stages. In the design stage, we introduce original guidelines to develop RF models that are compatible by design with the specifications of the target network hardware, by adapting (i) feature engineering, (ii) hyper-parametrization and (iii) memory management to the specificity of the switch architecture. Concerning model operation, we devise a comprehensive system for flow management suited for programmable ASICs, which performs (i) tracking of the relevant flows, (ii) computing, storing and updating of the FL features, and (iii) looping of inference results to the control plane to optimize the switch memory usage.
- Our framework can support diverse strategies to map RF models onto Protocol Independent Switch Architecture (PISA) pipelines. To prove our point and understand the impact of the RF mapping strategy on the final performance, we implement a default version of `Flowrest` based on the state-of-the-art RF mapping proposed by Planter [23], and then produce variants based on three other RF mapping schemes proposed in the literature.
- We implement `Flowrest` into a real-world Intel Tofino switch using the P4 language, alongside 5 benchmarks representative of existing RF-based in-switch inference solutions in the literature. We make our source code publicly available, so as to foster the reproducibility of our results and reduce the current significant barriers in accessing hardware-ready code for in-switch inference.
- Extensive tests on an experimental testbed and in the presence of high-bandwidth background traffic at 40 Gbps demonstrate the consistent gains achieved by `Flowrest` over existing solutions across a variety of use cases based on real-world measurement data. Namely, `Flowrest` improves accuracy by over 10 percentage points on average with respect to the second-best solution, while retaining sub-microsecond delay and similar usage of key resources like Ternary Content Addressable Memory (TCAM).

II. Background on line-rate inference

We offer a concise yet comprehensive review of existing proposals for line-rate inference that span the entire user plane

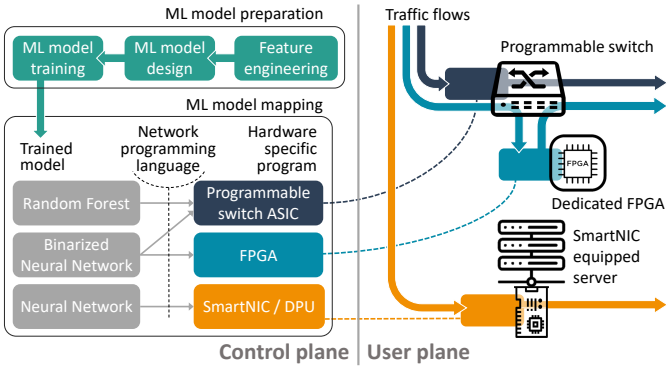


Figure 1: Summary of the approaches for line-rate inference.

in Section II-A and then shift our focus to in-switch methods in Section II-B. Given our study's emphasis on pure user-plane implementations of FL models: hence we do not explore, for example, non-ML approaches which use traffic filters [24], modular defence primitives [25], or user-plane scanners [26] to achieve high-speed inference for different use cases. Similarly, hybrid ML-based approaches that divide the inference process across planes [15], [27]–[30], and objections that target such hybrid approaches [31] are not relevant to our work.

A. Machine learning in the user plane

Figure 1 summarizes the different workflows utilized for line-rate inference to date. Due to the computational constraints of programmable network hardware, all existing approaches presume that the entire model preparation, which includes computationally intensive model hyper-parameter optimization and training, occurs offline in the control plane using dedicated GPU or TPU resources.

After training, and still in the control plane, models undergo encoding for deployment in the target user-plane equipment using a network programming language. Next, we review previous studies based on their hardware target, concurrently examining the particular ML models they deploy.

Programmable switch ASICs. The predominant effort thus far has been directed toward in-switch inference. As illustrated in Figure 1, in this scenario, the FL model is completely integrated into a single programmable switch by implementing it into the switch ASIC, such that line-rate packet processing is enabled by utilizing only the switch's resources. In this case, Decision Tree (DT) and Random Forest (RF) models are the most popular choices of ML architectures, thanks to their relatively low complexity and inherent suitability for mapping to the ASIC pipeline. Since we also employ this approach for our solution, we delve into a detailed discussion of current in-switch implementations of DTs and RFs in Section II-B.

Several other ML models such as Support Vector Machines (SVM), Naive Bayes, K-Means, XGBoost, and Isolation Forest have been also considered for in-switch inference. Yet, it has been determined that DTs and RFs ultimately attain superior scalability and performance [23], [32]. The same applies to efforts directed at more complex models, like Neural Networks (NN) [23], [33], potentially in the form of a Binarized Neural Network (BNN) [34] that rely on $+1/-1$ weights and sign

activations [35]. Even simplified versions of neural networks pose challenges when deployed into standard programmable switches: for instance, a very simple BNN with two layers of 64 and 32 neurons already depletes the resources of an Intel Tofino ASIC [34], and ultimately leads to poor inference performance due to its excessive simplicity. To address this, recent endeavours consider possible distributed deployments of neural networks with several switch nodes sharing the burden of complex models [33], yet such approaches are still in their infancy and introduce a need for multi-switch inference.

SmartNICs. The inherent constraints of programmable switches in accommodating neural networks have prompted the investigation of alternative hardware in the user plane for implementing such models. A recent work, N3IC [36], introduces a complete method for integrating BNNs on SmartNICs. Figure 1 illustrates the process by which N3IC maps the FL model onto SmartNICs situated in a middlebox or server.

The SmartNIC compute capabilities mitigate the limitations of programmable switches, allowing N3IC to implement a 3-layer BNN that functions at line rate while using only a small portion of the available resources. However, SmartNICs are deployed in network appliances such as load balancers, traffic classifiers, or security middleboxes, which are physically located in dedicated hosts within the network datacenter. Also, a single SmartNIC is priced similarly to a production-grade programmable switch but has fewer ports (1–4 versus 16–64), making a widespread SmartNIC-based solution significantly less cost-efficient on a per-port basis [37]. Due to the limited deployment flexibility and high costs above, approaches like N3IC can provide line-rate inference only at specific locations within the network, and not pervasively across the transport domain as is instead enabled by in-switch solutions.

FPGA-enhanced switches. The incorporation into switches of dedicated FPGAs to execute complex NN models has been also proposed. Taurus [38] employs a specialized accelerator featuring a pipelined Single Instruction Multiple Data (SIMD) parallelism to implement NNs through MapReduce operations. Taurus-enhanced switches delegate the per-packet inference process to external hardware, facilitating the deployment of robust ML models in user planes. The limitation of this approach is evidently rooted in the considerable added costs and technical complexity of associating FPGAs to each programmable network equipment. Implementing such a strategy at scale would necessitate a revision of the user plane design and the deployment of significant specialized hardware alongside already expensive programmable switches and SmartNICs.

B. In-switch tree-based inference

Our emphasis is on in-switch inference using DT and RF models. As explained earlier, executing ML models on standalone programmable switch ASICs allows for the most pervasive user-plane deployment without incurring additional hardware costs; and, tree-based models stand out as the most effective solutions in these highly constrained environments.

Table I provides a summary of prior studies that have explored in-switch tree-based inference, categorized into three dimensions: design methodology, scalability to complex models and use cases, and suitability for practical environments. We next delve into the specifics of prior solutions and compare them to our approach across the three dimensions above.

Design. Regarding model design, the main dichotomy is between performing inference at packet level or flow level. In PL design, models are fed with features extracted independently from the headers of each packet. The PL approach makes implementation easier, as feature extraction can be directly realized via the native header parsing functionalities of programmable switches. Yet, it also has notable drawbacks. First, PL models lack access to features calculated over multiple packets of the flow, such as inter-arrival times or packet size statistics, which play an important role in accurately solving complex network problems [56]–[60]. Second, PL models must be executed on each and every packet traversing the switch, which may incur in higher energy consumption due to the more frequent access to power-hungry TCAM [61].

On the other hand, FL inference utilizes multi-packet features computed over the first few packets of a flow and then applies the result to all subsequent packets of the same flow. It thus has the potential to remove the limitations of PL solutions. Nevertheless, the FL approach also introduces substantial realization challenges, such as the need to store stateful FL features or to manage classified flows. As a result, and in spite of its advantages, only a subset of the solutions in the literature support FL functionalities, as shown in Table I.

Among all FL inference approaches proposed to date, ours is the first to propose a hardware-tailored model configuration. pForest [40] and SwitchTree [41] have incorporated feature selection into their operation, but it functions as a traditional model preparation stage and is entirely indifferent to in-switch requirements. Although pForest [40] introduced a feature compression method to conserve switch resources, its practical viability is questionable[1]. Friday et al [54] made an early attempt at considering the constraints of programmable switches in the RF modelling process. They employ Gray coding to the feature table codes, enabling the reduction of TCAM consumption. However, this is more of a post-training optimization, performed after the RF model has been generated. In contrast to prior proposals, `Flowrest` is the first to customize feature engineering and hyper-parameter tuning specifically for the underlying user-plane operation. As elaborated in Section IV-A, this guarantees the compatibility of trained models with off-the-shelf programmable equipment already at the RF design stage.

Scalability. We examine scalability from two angles, *i.e.*, (i) support for RF models with multiple trees and (ii) the maximum achievable depth of each RF tree.

Regarding RF support, the majority of solutions facilitate multi-tree RF designs. However, recent approaches like NERDS [43], Soter [52], pHeavy [45], and Mousika [46] are constrained to single-tree models, *e.g.*, DTs or Binary DTs (BDT). This limitation is a notable drawback, as RFs gener-

[1]The proposed bit-level compression of features in pForest [40] enables the conservation of a few bits in the representation of each feature. However, it is worth noting that commercial programmable switch ASICs only support byte-level memory allocation, rendering the majority of bit-level optimizations meaningless once the model is deployed in hardware.

Solution	Design		Scalability		Practicality			
	HW-tailored configuration	Flow-level support	Forest support	Unrestricted tree depth	General purpose	Hardware implementation	Resource usage analysis	Open code
Ilsy [32], [39]			✓	✓	✓	✓		
pForest [40]		✓	✓		✓	✓		
SwitchTree [41]		✓	✓		✓			✓
Planter [23], [42]			✓	✓	✓	✓		
NERDS [43], [44]		✓			✓			✓
pHeavy [45]		✓				✓		
Mousika [46], [47]				✓	✓	✓	✓	✓
BACKORDERS [48]		✓	✓					
Netpixel [49]				✓				✓
Henna [50]			✓	✓	✓	✓	✓	✓
Bütün et al. [51]			✓	✓		✓	✓	✓
Soter [52]						✓		✓
INC [53]		✓		✓		✓		
Friday et al. [54]		✓	✓	✓		✓		
NetBeacon [55]		✓	✓	✓	✓	✓	✓	✓
`Flowrest`	✓	✓	✓	✓	✓	✓	✓	✓

Table I: Comparative summary of prior solutions for in-switch inference and `Flowrest`. Columns refer to (i) adoption of ML modelling and experiment configuration approaches that are tailored to the switch hardware requirements by design, (ii) support for flow-level inference, (iii) support for Random Forests composed of multiple Decision Trees, (iv) lack of structural constraints to the depth of the trees, (v) applicability to general inference problems opposed to solving a dedicated task only, (vi) implementation and experimental evaluation with a real-world hardware platform, (vii) complete analysis of switch resource consumption based on memory types, ($viii$) availability of open-source code.

alize DTs and are widely acknowledged to possess superior learning capabilities, particularly in more challenging tasks.

Existing solutions utilize various methods to map DTs or RFs into the PISA pipeline employed by most modern programmable user planes. Several of these methods impose structural constraints on the maximum attainable tree depth. For example, the mapping strategy employed by pForest [40], later adopted by SwitchTree [41] and in a modified form by BACKORDERS [48], maps each tree level to one Match-Action Unit (MAU) stage of the PISA pipeline. Thus, the tree depth is limited by the relatively small number[2] of MAUs in commercial switch ASICs. Soter [52] also suffers from this limitation as it again employs 1 MAU stage per level of the tree, differing from pForest [40] only by the use of TCAM range matches to compare feature thresholds. In addition, NERDS [43] and pHeavy [45] use mappings in which conditional statements and/or distinct trees are linked to individual MAU stages and need to be executed sequentially. This sequential operation across a restricted number of MAUs imposes a natural limitation on the model complexity. Mousika [46], [47] takes a more radical approach by generating a binary DT (BDT) from the original model and mapping only the BDT to the switch pipeline. While this brings about a more compact representation of the model, binarization often results in loss in accuracy as shown in Section VII-B.

Other approaches use tree encodings that dissociate tree levels from MAUs, eliminating the systemic constraints mentioned earlier. The mapping approach initially introduced by Ilsy [32] and subsequently extended by Planter [42] represents the state of the art, as it allows integrating in real-world hardware multiple trees with a depth constrained by the switch memory only. Most recent works either re-use this mapping scheme [49]–[51] or design new strategies closely related to the former [53]–[55]. Among these, the strategies proposed in INC [53] and later extended in Friday et al. [54] are specifically tailored to binary classifiers where only leaf nodes with a positive class result are encoded., which limits their application use cases. Hence, while `Flowrest` has the flexibility to adopt various methods for mapping RF model into PISA pipelines, it currently uses a custom implementation of the mapping approach proposed by Planter [42], as detailed in Section IV-B, to guarantee scalable multi-tree support.

Practicality. Either software or hardware targets can be employed for in-switch inference implementations using the same network programming language, such as P4. When the solution is developed for software, it can only be assessed through emulation *e.g.*, by executing it on the widely used *bmv2* target within a Mininet environment. While valuable for initial development and debugging, software implementations deviate significantly from production-level hardware targets, such as Intel Tofino ASICs. Disparities extend beyond differences in throughput and latency: emulation masks many limitations of the real-world equipment and solutions exclusively tested in software often lack compatibility with actual programmable switches, need substantial re-design efforts to port them in hardware, and yield performance losses [62].

Hence, the primary characteristic determining the practical feasibility of a model is its potential to be implemented in real-world hardware. According to Table I, this is applicable to the latest versions of Ilsy [39] and Planter [23], along with Henna [50], Soter [52], and Bütün et al. [51], all of which only support PL inference. INC [53] and Friday et al. [54] also have hardware implementations but are only adapted for binary classification and are not directly extensible for general-purpose inference. Other recent contributions, such as pHeavy [45], Mousika [46] and NetBeacon [55], showcase hardware implementations, albeit limited to DTs, BDTs or multiple non-RF DTs. Also, we note that not all the existing solutions make their source code publicly available.

Finally, we provide commentary on two more facets in Table I. First, most of the proposed models are versatile and

[2]In Intel Tofino switches, first-generation chips have 12 Match-Action Unit (MAU) stages, while second-generation chips have 20 MAU stages. In the context of FL inference, certain stages need to be allocated for computing flow identifiers and register indices, handling stateful features, and implementing tree leaves. This restricts the maximum tree depth, *e.g.*, pForest [40] is bounded in hardware to relatively shallow trees with a depth of 4.

applicable to diverse inference tasks, however, pHeavy [45] is dedicated to a specific objective, specifically, the binary identification of heavy flows; INC [53] and Friday et al [54] are also specifically dedicated to botnet propagation and ransomware attack detection, respectively. Second, aside from the PL models of Mousika [46], Henna [50], and Bütün et al. [51], alongside the hybrid model of NetBeacon [55], other prior works do not explicitly outline the requirements of the models in terms of switch resource usage.

Progress beyond the state of the art. As shown in Table I, `Flowrest` is the only solution to tick all boxes. Our open-source solution supports general-purpose FL inference via RFs that are hardware-compliant by design. A thorough comparative evaluation with representative models from Table I is provided in Section VI and demonstrates how the innovations set forth by `Flowrest` ultimately result in better and more consistent performance at no resource usage cost.

III. Flowrest in a Nutshell

`Flowrest` is a comprehensive system for the classification of flows in the user plane, whose overall operation is illustrated in Figure 2. The system has two main logical parts, deployed in the control plane and user plane, respectively. In the control plane, the ML model is prepared and runtime parameters, crucial for the system scalability, are selected; these operations happen offline. In the user plane, three main functional elements reside: (*i*) the *Inference-aware forwarding*, designed to ensure the coexistence of the inference logic with the normal forwarding function of the switch; (*ii*) the *Flow management*, where new flows are tracked, and FL features are calculated, stored, and updated; (*iii*) the *RF model inference*, classifying flows of packets at line rate. At runtime, an online controller interacts with the user plane upon flow classification, consolidating forwarding rules and freeing switch memory. In Figure 2, the offline preparation steps are highlighted with encircled numbers ①–⑥, while the operations performed in the data plane are tagged with encircled letters ⓐ–ⓙ.

In step ①, a grid search is performed with hyperparameters and features, tailored to the inherent hardware constraints, as detailed in Section IV-A, and the best model is selected in ②. In ③, the runtime access of the flows to the switch memory is simulated to dimension the flow management registers and define the timeout for flow expiration, as described in Section IV-C. These two parameters, enforced in ⑤, are crucial to guarantee *low flow collisions* without performance degradation. In ⑥, the model is injected into the data plane.

The data plane operation then starts in ⓐ, where the incoming packets traverse the inference-aware forwarding block, as detailed in Sections V-A and V-B. Packets that are not a target of inference are forwarded in ⓑ with no further action. The packets that have to be classified go instead into the flow management block in ⓒ.

To classify flows, `Flowrest` exploits FL features calculated over the first n packets of each flow. Each new flow traversing the switch is then tracked in ⓓ, with a mechanism based on the hashing of its 5-tuple that assigns a specific memory slot, as detailed in V-C. In case the new flow finds its slot occupied

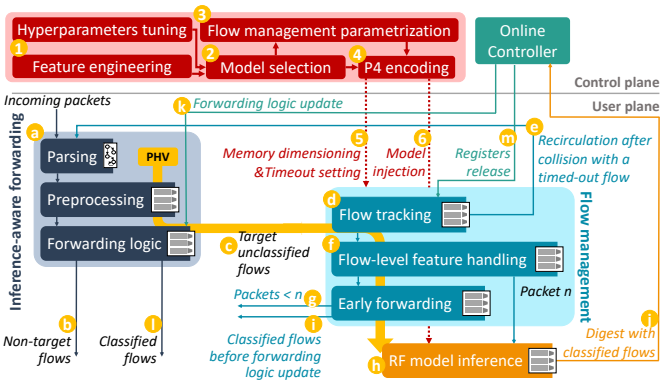


Figure 2: Overview of the system proposed for `Flowrest`.

by another flow, a hash collision event occurs, and the new flow cannot be tracked. To minimize the chances of missing the new flow after a collision, we introduce a mechanism that releases the memory slot if the flow occupying it has timed out. In such a case, depicted in ⓔ, the new flow packet is recirculated to the beginning of the pipeline, as Tofino Native Architecture (TNA) does not allow access to the same register more than once per packet. In ⓕ, the FL features are updated after each new packet of the same flow arrives. The packets arriving before the n-th are forwarded with standard rules in ⓖ, as they do not have a class yet. In ⓗ, instead, the n-th packet goes to the inference module to be classified.

After the classification, two actions are taken. The first one is to update the early forwarding element with the class so that all the following packets can be forwarded in ⓘ, avoiding reclassification. The second one, depicted in ⓙ, is sending a digest to the controller with the class information, as detailed in V-C. Upon receiving the digest, the controller can consolidate the class of the flow by updating the forwarding logic in ⓚ, so that newly arriving packets of the flow can be forwarded according to a predefined rule in ⓛ. The controller also releases the registers allocated for the flow in ⓜ.

IV. Hardware-Tailored Modeling

A unique characteristic of `Flowrest` is the fact that *we consider hardware constraints* already during the preparation of the FL model and in the choice of experimental parameters like switch memory dimensions and flow timeout thresholds. This facet of `Flowrest` is enabled through the stages of model preparation and the switch configuration simulator depicted in Figure 2 by the steps ①–③ which are executed offline in the control plane. Next, we detail these steps.

A. Switch-tailored model design

Our hardware-tailored modeling approach is derived from how flow management and RF inference mapping are carried out. This process guarantees that the trained RFs are *inherently* compatible with the programmable switch. Like most prior studies, we utilize the widely used Scikit-Learn libraries [63] during the model preparation phase. However, we customize the feature engineering and hyperparameter selection to ensure seamless integration into real-world equipment. Our entire FL pipeline is structured around the following steps.

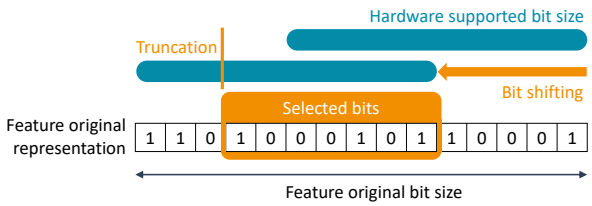


Figure 3: Hardware-aware tuning of feature representation.

Feature extraction. At this stage, we compute both PL and FL features for the training data after extracting packet header information from historical pcap traces via Tshark [64]. Although we can collect any set of features, we use the TCP/UDP source and destination ports, packet length, and TCP flags (ACK, SYN, PUSH, ECE, RESET, FIN) as stateless PL features for all the experiments in this paper and we utilize statistics on the counters of the TCP flags mentioned above, packet length (minimum, maximum, total, and mean), inter-arrival time (minimum, maximum, and mean), and flow duration as stateful FL features.

A crucial point to note is that, in contrast to some prior studies [23], *we refrain from incorporating IP and MAC source or destination addresses as features*. This decision is grounded in the fact that using these end host identifiers can unnaturally boost the model's performance in the majority of scenarios analyzed in existing literature, as well as in our own work. These use cases rely on measurement data collected in limited scenarios. An adequately complex model, when provided with IP or MAC addresses during training, has the ability to learn the association between end hosts and the target classes (e.g., send traffic belonging to a specific service, or show malicious behaviours). Thus, in tests where the same end hosts are repeatedly used, the inference task becomes artificially straightforward as it is seen in most of the use cases adopted by the works depicted in Section II-B. However, once deployed in real-world scenarios, the model would fail spectacularly because it would not encounter the same hosts it had learned to recognize. By omitting identifiers, we compel the model to perform the task relying on the inherent characteristics of the traffic, so we ensure that the model can generalize its learning to hosts it has not encountered before.

Feature engineering. We adhere to traditional feature selection methods, where we first train a RF model using all available features and rank them based on Mean Decrease in Impurity (MDI). Subsequently, we train additional models by gradually adding features to identify the smallest subset that accomplishes performance comparable to or better than the model trained with all features. Our feature engineering approach is unique since *we tailor the feature representations to match the hardware specifications*. Specifically, state-of-the-art RF mapping techniques employ range matches to compare feature values against thresholds from the model, as detailed in Section IV-B. The size of these range matches is limited to a maximum number of bits in real-world equipment.[3] Although this limitation does not impact PL features, which typically require only a few bits, it does affect FL features. Specifically, time-related features, often represented with floating-point precision by default, are influenced by this constraint.

To address this challenge, `Flowrest` fine-tunes features that exceed the size limit by (i) shifting their binary representation and (ii) truncating it to fit within an ideal bit length smaller than the limit. This process, depicted in Figure 3, occurs during training through an exhaustive search to find the optimal combination of shift and truncation for each feature. The outcome can be implemented in hardware during the FL feature handling phase, as it involves only simple binary operations. The fine-tuning process guarantees that all comparisons utilized by the RF model are inherently compatible with the capabilities of the target programmable switch.

Model design. We perform a traditional grid search to determine the optimal values of the maximum number of trees (t), the maximum depth of each tree (d), and the maximum number of tree leaves (l) jointly with the feature selection mentioned earlier. Thus, the feature ranking and processing outlined previously are applied to each RF model with specific hyperparameters t, d, and l. We evaluate models across all combinations of (t, d, l). The size of the model fundamentally depends on the complexity of the specific use case: for example, for the use cases in our experiments, we consider d in the range $[3, 20]$, t in the range $[1, 5]$, and l in multiples of the width of a TCAM block, enabling us to also *customize the selection of hyperparameters in the RF model to match the specifications of the underlying hardware*.

State-of-the-art RF mapping techniques, like Planter [65] and Mousika [46], utilize ternary matches on codewords and corresponding masks for the final classification. This means that the length of codewords must adhere to the maximum ternary matching size permitted by the switch's Ternary Content Addressable Memory (TCAM). During the design phase, we ensure that the aforementioned constraint is satisfied by the model, by leveraging the fact that each tree in the RF generates a single codeword, where each bit represents one node of the RF (excluding leaves). In a full binary tree with a total of $N-1$ nodes, there are always N leaves. By limiting the number of allowed leaves to match the maximum ternary matching size in the programmable switch, we can control the length of the codeword. This control is implemented by selecting the best nodes in each tree based on the relative reduction in impurity [66] until the desired number of leaves is reached. Note that restricting the number of leaves does not limit the tree depth, which can extend up to $d = N$, allowing the model to maintain its flexibility.

Once the full space of hardware-feasible models is explored, we run a selection algorithm inspired by that originally proposed by Jewel [67]. We pick the best model for in-switch implementation based on the following performance score:

$$\alpha \frac{1}{2} \left(\mathrm{F1}_{\text{macro}} + \mathrm{F1}_{\text{weighted}} \right) + (1-\alpha)(1-\rho), \qquad (1)$$

where $\mathrm{F1}_{\text{macro}}$ and $\mathrm{F1}_{\text{weighted}}$ are two metrics of inference accuracy based on the F1 score that are derivable from the training data as detailed in Section VI-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

[3]The exact maximum size depends on the equipment and is confidential.

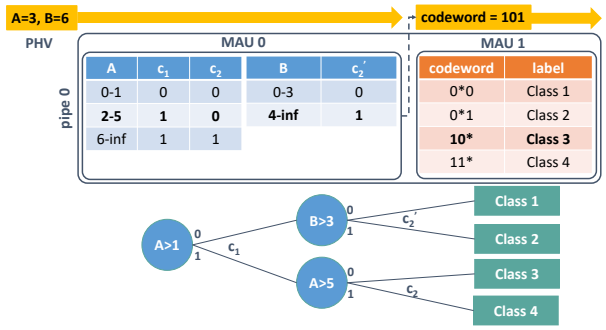


Figure 4: Overview of the Planter RF model mapping to MAUs adopted by `Flowrest`. Each feature employed by the RF is mapped to one MAU, and different trees of the forest share match-action tables. The compiler optimizes memory usage by storing multiple features of a same tree in the same MAU.

parameter used to weight the two contributions of accuracy and resource usage. In our experiments, we use the fraction of consumed TCAM for ρ, as it is arguably the most limited and costly resource in our target hardware. Additionally, we set α to 0.5 to achieve a fair balance between the two components.

B. RF model mapping and inference

The framework designed for `Flowrest` can adapt to any hardware-feasible model or mapping strategy without requiring modifications to the previously outlined workflow stages. To the best of our knowledge, our framework is the first solution for FL in-switch inference that offers effortless integration of various model mapping strategies. To create a comprehensive solution for in-switch FL RF-based inference, we reproduce a custom version of the mapping method introduced by Planter [42] and integrate it into our framework. We opt for this choice because, as elaborated in Section II, it represents a state-of-the-art mapping strategy guaranteeing scalable multi-tree support. Below, we provide a summary of how the Planter RF mapping operates.

The distinguishing element of this mapping approach is that specific match-action operations are assigned to individual features. This means that each feature is processed within a single MAU, even if it is utilized across multiple levels of different trees. In this approach, it is necessary to handle all decisions made based on the same feature across all nodes of all trees in the RF model all together. The toy example, depicted in Figure 4, illustrates this concept. The diagram is specific to a single tree, but it can be readily extended to apply to the forest scenario. Initially, value ranges are calculated for each feature to encompass all the thresholds that the feature may come across in the RF. These ranges are then stored in the match section of a MAU. For example, as shown in Figure 4, the feature A is compared to the thresholds 1 and 5 at nodes C1 and C2, respectively. Consequently, the match-action table corresponding to feature A includes three distinct value ranges: values less than or equal to 1, values between 2 and 5, and values 6 and above. Then, for every node using the feature, each value range above is associated with a binary outcome: true (1) or false (0), through the action stage. For example, as depicted in Figure 4, the first decision C1 (which checks if $A > 1$) is set to 0 for the match row where $A \in [0, 1]$, and to 1 for the two rows where A is strictly greater than one, *i.e.*, $A \in [2, 5]$ and $A \in [6, \infty)$. Instead, the second decision C2 is set to 1 only in the last case, $A \in [6, \infty)$. Through the match-action operation, a code 10 is assigned for feature A to a packet with A=3, and the code is stored in the PHV.

A full *path code* for the packet is constructed by iterating this procedure for every feature and concatenating the action outcomes in the PHV. In the final MAU, every tree in the forest has a specific match-action sequence, where the path codes are matched against predefined sequences that outline all potential paths from the tree's root to its leaves. It enables the packet to be matched with a specific path, along with its corresponding label, which is then stored in the PHV.

An important note is that not all potential path codes are feasible. This is because each bit corresponds to a specific node, and a path only encompasses a subset of these nodes. Hence, we utilize wildcards in the final match stage to represent bits (*i.e.*, nodes) whose values are irrelevant to the inference decision because they are not part of the path. As a result, in the final (MAU), the mapping significantly depends on Ternary Content Addressable Memory (TCAM), which is necessary for handling non-exact matches involving feature value ranges and wildcards.

Other mapping schemes considered include those proposed in pForest [40], Soter [52], and Mousika [46]. pForest maps each level of a DT to one MAU stage of the switch pipeline as described in Section II-B, limiting the maximum depth of trees to the number of available stages on the switch ASIC. At each DT level, the feature values are compared to the thresholds in the tree nodes. The result of the comparison determines that path that is followed to reach the corresponding leaf node. As this comparison is not trivial to achieve in hardware, it often requires additional stages to implement, further hampering scalability and limiting most deployments to the *bmv2* target.

Soter [52] employs a similar approach as pForest, mapping each level of a tree to one MAU stage but overcomes the need for additional stages for logical operations by using TCAM-based range matches to compare the feature thresholds and feature values at tree nodes. This introduces TCAM consumption but ensures that for trees of maximum depth N, it requires $N+1$ stages, with the extra stage used for initializing the tree root nodes. The available TCAM at each MAU stage is also a constraint since the same levels of different trees have to share those resources. This impacts scalability.

Mousika [46] takes a more radical approach to tackling model complexity by employing knowledge distillation to generate a compact student BDT from a complex teacher model. The BDT is then mapped to the switch pipeline, where in the first step a code word is generated from the binary feature values, and then matched onto a code table in the second phase to assign a classification result. While the BDT is mostly compact, it could sometimes grow into very large memory-hungry sizes as will be shown in Section VII-C. Distillation could also result in loss of accuracy.

C. Switch configuration simulator

The flow management portion of the `Flowrest` system, and more precisely the flow tracking in step ⓔ of Figure 2, store stateful information about individual flows traversing the switch. The dimensioning and handling of the SRAM registers used to implement such state is not trivial: over-dimensioning leads to unnecessary consumption of the already scarce switch resources, whereas under-dimensioning leads to the impossibility of tracking and classifying all flows; similarly, discarding inactive flows too early may delay their classification, while waiting too long again results in unavailable registers and unclassified flows. The problem is aggravated by the fact that the correct memory and timeout settings are specific to the inference tasks and there is no one-size-fits-all solution.

Searching the space of flow management configurations via experiments in hardware is not viable, as it would require a huge engineering effort of coding, deploying and testing many versions of `Flowrest` in real-world programmable switches. It thus makes sense to tackle the problem offline at the system design stage, leveraging the much larger compute resources of the control plane. Our approach is that of developing a dedicated simulator, capable of closely mimicking the capabilities and behavior of an Intel Tofino ASICs in terms of its flow management logic, and to employ it for swift and exhaustive searches of the parameter space prior to the implementation of the model in the actual switch, as follows.

Simulator workflow. Whenever a packet enters the simulator, if the flow is not classified, the simulator computes a 16-bit *flow index* as a CRC16 hash of the 5-tuple, used to identify the entry where the current flow is stored in the flow management table, by using `crcmod` function of Python. We used the same constant hash parameters with the predefined `HashAlgorithm_t.CRC16` function adopted by the Intel Tofino Native Architecture (TNA). The simulator uses the B significant bits of the generated hash value as the flow index to locate flows, where B is the *bit-width* of the flow management table, *i.e.*, $\log_2$ of the total register entries in the table.

If a different flow is stored at the flow index, a *flow collision* occurs. It may be the case that the flow in the flow management table has *timed out*, *i.e.*, the time elapsed since the arrival of the last packet of the stored flow is above a user-defined threshold. In this case, the simulator mimics the *recirculation* of the packet encountering a collision because of a timed-out flow by adding a delay of few nanoseconds before purging the data from the related entry and replacing it with the newly arrived flow. If instead the earlier flow has not timed out, the colliding packet is simply forwarded and not classified.

In the case that the flow index of the table is empty, the flow occupies the entry. The simulator initializes all data to be stored in the table, which encompasses the stateful features needed by the FL RF model, the count of recorded packets in the flow, and a timestamp of the arrival of the last packet used for deciding whether the flow is timed-out.

As a final option, if the flow index of the table is already used by the same flow, the simulator updates the flow management table entry with the current packet metadata.

When the flow management processes the n-th packet of a not yet classified flow, the simulator decides the class of the flow by feeding the RF model generated offline with the FL features stored in the table and PL features needed by the model. The simulator then mimics the digest informing the controller about the classification result by adding a 10 ms delay before removing all the information from the index belonging to the just classified flow and updating the status of the flow as classified. As such, subsequent packets of the flow will directly be forwarded without any need for classification. The 10 ms delay between data plane and control plane is determined via an exhaustive search in the simulator that returned the number of classified flows closest to the one obtained by the experiment in the switch.

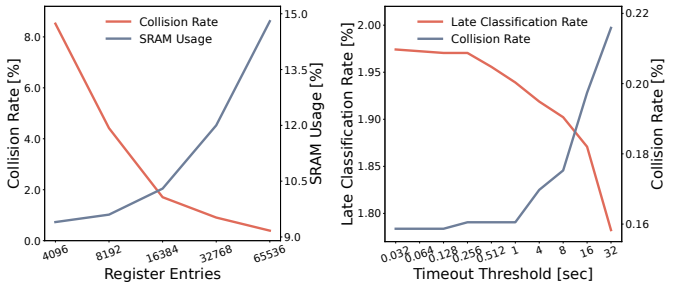


Figure 5: Collision rate calculated by the simulator under varying number of entries used in the flow management tables (left) and percentage of late classified flows returned by the simulator based on the timeout threshold (right), for the ToN-IoT use case presented in Section VI-B.

Simulation-driven flow management setup. For each target inference use case, the simulator is used to dimension the flow management registers allocated in the switch. Specifically, exhaustive yet fast simulations allow fully characterizing the relationship between the number of register entries and the collision rate so as to select the optimal trade-off of SRAM consumption and flow collisions. Similarly, the simulator mimics the timeout mechanism and can thus inform the choice of the optimal timeout threshold used to decide whether to discard a flow stored in the management registers without causing early dropping of flow status prior to classification.

Visual examples of (i) collision rate versus register entries and (ii) fraction of late classified flows due to early dropping versus timeout threshold, obtained by the simulator for the ToN-IoT use case in Section VI-B, are shown in Figure 5.

V. Practical flow-level in-switch inference

The core of `Flowrest` is the in-switch component that comprises the steps ⓐ–ⓙ in Figure 2, which together enable line rate FL inference as detailed next.

A. Parsing

Packets arriving at the switch undergo processing through a PISA pipeline, which consists of a parser followed by a series of Match-Action Units (MAUs). The parser retrieves metadata from the headers of individual packets. We configure the parser to also extract PL information specifically pertinent to the intended inference task in addition to the metadata serving only forwarding decisions in standard switch operations. The information is stored in the Packet Header Vector (PHV),

which consists of containers carrying relevant raw header fields (*e.g.*, the source port) and collected metadata (*e.g.*, the packet arrival timestamp) across the entire MAU pipeline.

B. Inference-aware forwarding

Packets then proceed through the MAU stages that perform the standard switch functionalities by pre-processing the packet metadata for legacy forwarding, but also implement advanced forwarding logic. Specifically, in `Flowrest` *we directly incorporate in the forwarding logic configurations that are associated to the inference process*, as follows.

- First, at the time when the RF model is programmed in the switch, the controller also configures the forwarding logic so that only the target traffic is filtered for inference. The target may be a certain range of source IP addresses that is thought to be involved in malicious activities, or a group of protocols known to facilitate the target traffic. By incorporating target traffic into the forwarding logic, we designate packets for in-switch inference. If no notion of target traffic is associated to the considered inference task, then all traffic flows undergo the classification process.
- Second, target flows that have already been classified are added to the forwarding logic and associated to dedicated rules. In this way, classified target traffic does not undergo the inference step anymore rather is dropped, tagged or forwarded to specific ports or destinations immediately in the forwarding stage.

Hence, the forwarding logic operates directly on flows that do not undergo inference or for which RF results are accessible. Figure 2 portrays how regular traffic follows legacy forwarding rules to reach egress ports, whereas flows flagged as adversarial by the inference process are immediately dropped.

To implement the inference-aware forwarding above, the controller is informed about the classification results and utilizes them to set up the forwarding pipeline. As depicted in Figure 2, once a flow has been classified by the RF model the switch is programmed to notify the controller by sending a *digest*, *i.e.*, a compact message specifically designed for communication with the control plane. The `Flowrest` digest includes the unique flow identifier introduced in Section V-C, the corresponding inference result (*i.e.*, the flow class), and the flow management table location. By using this information, the controller updates the forwarding rules.

An inference-aware forwarding has significant benefits on the user-plane inference process. On the one hand, packets belonging to flows that have been classified or do not require classification only pass through the standard forwarding logic, which makes processing of many flows much faster. On the other hand, utilizing forwarding tables for already classified flows allows early-purging such flows from the resource-limited flow management registers, as explained in Section V-C, and improve the scalability of the solution. By introducing an inference-aware forwarding, `Flowrest` goes beyond current user-plane FL classifiers that do not integrate the control plane in their operation.

It is worth noting that the integration with the control plane above does not curb the concept of a pure user-plane

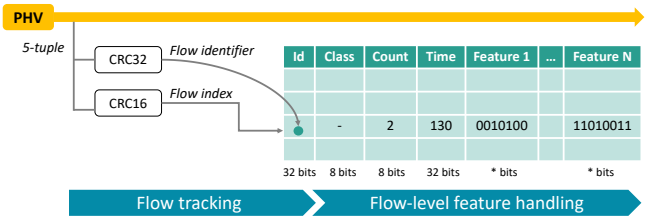


Figure 6: Flow management table and its associated stages.

inference that underpins `Flowrest`. While the reconfiguration of the forwarding logic by the controller takes milliseconds or higher, decisions for packets arriving immediately after a flow is classified are entirely made in the switch via the flow management routine later presented in Section V-C; this occurs until the forwarding logic is updated. This system guarantees continuous inference at line rate and with very low latency, either in the initial flow management MAUs or in the forwarding MAUs through a closed loop with the controller.

Finally, in operational network environments traffic conditions could change significantly over time, up to the point that the deployed model can no longer achieve high accuracy. This phenomenon, known as model drift, affects not only in-network solutions but all machine learning models that run in systems with time-varying properties [68]. Addressing model drift requires techniques like incremental model updates, automatic labelling, and model adaptation, which have been leveraged in recent works like LINC [69] and CARAVAN [70]. These techniques are complementary to our work: they can be adopted to keep up-to-date the design of stateful FL models that can then be deployed in the user plane via `Flowrest`.

C. Flow management

Packets identified by the forwarding logic as inference targets and not directly classified by forwarding pipeline as per Section V-B above are directed to the flow management stage. This stage handles the computation, storage, and use of FL information. It comprises three distinct phases, outlined below, all relying on a same *flow management table* depicted in Figure 6. The table, stored in Static Random Access Memory (SRAM) as a set of registers, contains essential data for each flow undergoing inference: (i) the flow identifier, (ii) its class if already determined, and (iii) its stateful features.

Flow tracking. This phase is responsible for recognizing incoming flows and tracking their progress within the flow management table. Initially, a *flow identifier* is created for the packet, via a CRC32 hash checksum of the 5-tuple, which includes IP source and destination addresses, source and destination ports, and transport protocol identifier. A *flow index* is also calculated using a CRC16 hash of the same 5-tuple. This index is employed to locate the corresponding entry for the current flow within the flow management table.

As depicted in Figure 6, when accessing the flow management table using the flow index, the first step is to compare the flow identifier with the stored value in the table. If there is a match or the entry is empty, the process moves on to the next phase. However, in cases of mismatch, indicating a hash collision, stateful information cannot be stored if the stored

flow has not timed out, leading to the absence of inference for that particular flow or a postponed inference to subsequent packets. Flow index collisions are clearly undesirable occurrences that negatively affect the overall performance of the in-switch inference. Unfortunately, the restricted capacity of registers and the high volume of flows passing through a switch may lead to frequent incidents of such collisions.

Prior proposals for cascading hashes in emulated environments [40] are prohibitively expensive to implement in hardware due to the need for a dedicated MAU stage. Instead, we realized that we can reduce the number of collisions with the help of the integration with the forwarding logic described in Section V-B. Offloading the inference-based decision to the standard forwarding pipeline for a specific flow right after the inference allows us to eliminate the corresponding entry from the flow management table, which determines a significant reduction in collisions by restricting the the simultaneous presence of flows in the table.[4]

We also implement a timeout mechanism, which helps us to remove the timed-out flows from the flow management table. We check whether the flow located in the table has timed out whenever it causes a collision. If the time elapsed since the arrival of the last packet of the the flow in the table is above a threshold whose value is fine-tuned using the simulation approach presented in Section IV-C, we recirculate[5] the colliding packet to start storing the stateful features of its corresponding flow after removing the data of timed-out flow.

Flow-level feature handling. After packets complete the flow tracking phase, all relevant information is computed, updated, and stored directly in the flow management table. As depicted in Figure 6, in each entry, we store the flow's class (if available), the count of recorded packets in the flow, and a timestamp indicating the arrival of the last packet of the flow used in timeout mechanism as previously explained. The table also retains all the stateful features required by the particular FL RF model being used. In our experiments, the selected features always represent a subset of the comprehensive FL features outlined in Section IV-A.

Several crucial points are worth highlighting in this phase.

- From a technical point of view, simultaneously reading and updating all stateful variables in the flow management table is crucial, yet challenging in hardware. To accomplish this, we employ the Tofino Native Architecture (TNA) RegisterAction extern function, which leverages the Stateful Arithmetic Logic Unit (S-ALU) associated with the registers for mathematical and logical operations.
- The counter is used to implement an *early-flow detection* approach [71], where `Flowrest` conducts inference upon receiving the initial n packets of each flow. To minimize latency and resource consumption, this approach classifies network flows using only the first few packets, recognizing that many flows consist of a limited number of packets while some may be very long, making it impractical to wait for all packets to arrive before performing inference. Instead, by considering only the first n packets of flows, enough information can be captured in computed features for flow classification with high accuracy. This approach has been demonstrated in several prior non-user-plane works [58], [71]–[73]. It is also used in many other in-network ML solutions at flow-level *e.g.*, pForest [40], NetBeacon [55], Brain-on-Switch [74], SwitchTree [41], and pHeavy [45]. The RF model processes only the specific n-th packet in the flow sequence,[6] which is an advantage over PL inference that requires processing all packets. The selected n maximizes accuracy after a thorough evaluation of possible values.
- The class assigned by the RF model to a classified flow is immediately stored in the corresponding flow management table, and the controller is informed of the RF model result through a digest message, after which it updates the forwarding logic. Until the controller updates the forwarding logic and packets are forwarded directly based on their class as described in Section V-B, incoming packets belonging to the newly classified flow pass through the early forwarding phase discussed next.

Early forwarding. Early-flow detection enables `Flowrest` to promptly respond to the results of the inference process. When a packet reaches the flow management stage and is recognized as a packet of already classified flow, it is directed to the early forwarding stage. In this phase, we employ specific logic tailored for classified flows, making forwarding decisions based on the class information stored in the flow management table. For instance, as portrayed in Figure 2, any packets beyond the n-th packet in a flow identified as malicious are immediately dropped at this phase. As previously discussed in Section V-B, the early forwarding process operates in the flow management for every newly classified flow until the controller receives a notification about the flow class and updates the forwarding logic accordingly.

It is important to note that the early forwarding stage is also designed to manage packets *prior to* the n-th in each flow. Although these packets contribute to FL statistics during the feature handling stage, they arrive too early to be classified. Directing them to the early forwarding phase guarantees their processing, *e.g.*, following standard rules.

Inference. When the flow management processes the n-th packet of a target flow not yet classified, the FL features, in addition to the PL ones, are stored in the PHV. As portrayed in Figure 2, these features then pass through an RF model encoded in the rest of the MAU pipeline as detailed in Section IV-B and produce a classification result per tree. A voting table performs a majority vote on the tree outputs.

As anticipated, upon classification `Flowrest` sends information to the controller through packet digests. After a flow is classified, a digest is assembled in the ingress deparser, comprising the 5-tuple that identifies the flow, the register index where the flow's data is stored, and the classification result. The digest is then sent to the controller which upon receiving them performs two actions. First, it sends and update to the inference-aware forwarding table, to indicate that the flow has already been classified. Second, it performs a *memory*

[4]In our use cases, the collision probability is close to 0% with `Flowrest`.

[5]Accessing the same register twice when processing a packet is not possible.

[6]In our experiments, we use early-flow detection with n = 2, 3, or 4.

refresh resetting to 0 all stateful registers at the index of the classified flow, hence freeing memory for new incoming flows.

VI. Experimental setup

We implement `Flowrest` in an experimental platform and conduct tests for various user-plane inference tasks with different levels of complexity, while also validating the capability of `Flowrest` to accommodate diverse RF model mappings.

A. Hardware setup

We implement our solution in a testbed comprising three industry-grade Edgecore Wedge100BF-QS programmable switches, each equipped with an Intel Tofino BFN-T10-032Q chipset and 32 100GbE QSFP28 ports. The switches run Open Network Linux (ONL) as operating system and Intel's Software Development Environment (SDE) version 9.7.0. The testbed also includes two servers featuring AMD EPYC 24-core processors at 2.8GHz and 128GB of RAM, and QSFP28 interfaces, creating a complete 100-Gbps platform for testing. We develop a Python controller which we lodge in one of the servers. the controller binds to the switches via the Barefoot Runtime Interface (BRI) and configures them at the beginning of the experiments by loading the P4 program, activating ports, and loading table entries of the trained RF models. The controller also listens on this interface during experiments to collect digest messages from the switch, which contain classification results and statistics on important events like hash collisions. We inject traffic into the switches from a server replaying pcap traces via Tcpreplay [75] using 100-Gbps connections. We also inject background traffic into the switch at 40-Gbps using the Moongen [76] traffic generator.

The SDE comes with the Intel P4 Insight analysis tool [77] which provides a detailed analysis of compiled P4 programs in terms of how they map onto switch resources and how they consume them. In order to derive estimates of the latency of P4 programs we employ P4 Insight after having run experiments to empirically validate its accuracy, as detailed next.

The packet processing latency through the switch is mainly composed of the time spent in the ingress pipeline, the traffic manager, and the egress pipeline. In P4 Insight, latency information is provided in terms of clock cycles. The latency in units of time can be estimated by multiplying the frequency of Intel Tofino switches with the total clock cycles utilized in the ingress and egress pipelines of the switch. The latency estimated in P4 Insight covers ingress and egress control, which has a fixed value but does not account for the time spent in the traffic manager. To verify the reliability of the estimate above, we implement latency measurement mechanism in the switch for a subset of all use cases, by tracking the time it takes for a packet to travel from the ingress MAC to the egress parser. This latency calculation occurs periodically, once for every *X* packets, and the calculated latency values are stored in a dedicated register within the egress pipeline. In addition, a separate register implemented in the ingress pipeline keeps count of the total number of latency values collected. Once the register storing the latency values is full, a digest that contains latency statistics is sent to the controller, which can then refresh the memory for further measurements. The latency measured in the switch covers the ingress pipeline, traffic manager, and egress parser: since no components of `Flowrest` runs in egress, we do not include this time in our calculations.

The latency measured in the switch is approximately 98 nanoseconds higher than the latency calculated through P4 Insight clock cycle measurements across various use cases. This variance primarily stems from the inherent delay within the traffic manager. Due to the significant burden that the latency measurement mechanism brings to the switch, in our experiments we derive the latency in nanoseconds using the clock cycle-based latency provided by P4 Insight, and adjusting it a-posteriori using the measured offset above.

B. Use cases

We run on the experimental platform various user-plane inference tasks in (i) device identification, (ii) IoT bot classification, (iii) IoT cyberattack classification, (iv) service classification, and (v) anomaly detection. With `Flowrest`, we design, train, evaluate, and deploy a model for each well-defined use case. We do not design a one-size-fits-all model that can perform domain adaptation or out-of-distribution testing, which are out of our scope but are well investigated in offline scenarios [78]. To ensure that our models are not overfitted, we implement train-test splits for each use case. As such, the test data is always unseen to the model, verifying its generalizability to unknown samples within the scope of the use case. The datasets underpinning all use cases are publicly available and are detailed next.

UNSW-IoT [79] is a use case where traffic generated within a living lab emulates a smart environment during 6 months. The measurements are obtained from a testbed consisting of 28 different Internet of Things (IoT) devices (*e.g.*, cameras and smoke sensors) along with several non-IoT devices (*e.g.*, phones and laptops). The goal is to determine the specific type of IoT device behind each traffic flow by analyzing the statistical features of the data packets. We use 15 days of data for training and one day for testing. The dataset comprises a total of $19,871,591$ packets spread into $492,961$ flows.

IoT-23 [80] is an IoT dataset that provides researchers with a comprehensive dataset that includes real and labelled IoT malware infections, as well as benign IoT traffic, for training ML algorithms. It contains packets captured between 2018 and 2019 from infected IoT devices and real IoT devices such as Philips HUE LED smart home, Amazon Echo Home smart personal assistant and Somfy smart door lock in controlled environments. We use the IoT-23 dataset to design in IoT bot classification use case where we differentiate 4 benign traffic classes from 10 malicious traffic classes, generated by different bots. It is made up of $2,015,636$ packets which make up $239,182$ flows. We extract a portion of the dataset with all classes represented and employ a 75-25 train-test split ratio.

ToN-IoT [81] comprises benign traffic and 9 kinds of IoT cyberattacks from measurements in an experimental platform deployed using various virtual machines such as Windows, Linux, and Kali Linux operating system hosts, to manage the connections between the Cloud, Fog, and Edge network

domains. The measurements are carried out with seven IoT and industrial IoT (IIoT) devices, as well as some non-IoT devices. We use a 75-25 ratio for train-test splitting of the data which counts $5,843,211$ packets distributed into $240,800$ flows.

UNIBS-2009 [82], [83] revolves around service classification dataset generated on the edge router of the University of Brescia campus network through three consecutive working days. The data is captured from a set of 20 workstations and encompasses various types of traffic: web (HTTP/HTTPS), mail services (POP3, IMAP4, SMTP), peer-to-peer applications (BitTorrent, Edonkey), and other protocols (FTP, SSH). We train and test the models using a separate day of data each, to categorize flows into one of the 8 application categories. The data comprises $3,666,016$ packets and $40,338$ flows.

CICIDS2017 [84] is an anomaly detection dataset generated in a testbed at the University of Brunswick. Through five consecutive working days, measurements are collected from two networks: a victim network, designed as a secure infrastructure with computers running a daemon that generates benign traffic, and an attack network executing 7 types of malicious traffic. We exploit the data for Friday, made up of $3,839,575$ packets and $625,956$ flows. We use a split ratio of 75-25 for training and testing, to classify traffic flows as either benign or attack.

C. Benchmarks and levels of comparison

We compare `Flowrest` to various benchmarks and organize our evaluation into different categories of comparisons.

1) Benchmarks: To establish `Flowrest` as the new standard for in-switch FL inference, we compare it to 5 benchmarks.

Planter [42] is a pure packet-level RF that uses the efficient mapping described in Section IV-B; it is thus representative of the performance of recent proposals like Planter [23], IIsy [32], Henna [50], and Bütün et al. [51] which re-use this scheme. We reproduce this benchmark with the source code in [85].

Mousika [46]. We implement our models with the Mousika mapping using the publicly available code [86]. Following the workflow of the original model, we take each of the sets of selected features for both the PL and FL models, binarize them and then use them to train a teacher RF whose prediction probabilities are then used to train a student BDT. The resulting BDT is then mapped onto the switch pipeline.

Soter [52]. We make use of the publicly available code [87] as a starting point, and then extend it to enable (i) support for RFs, and (ii) FL inference with `Flowrest`. In the PL models, RF models with maximum depths of up to 10 can be supported, with the extra stage reserved for voting. In the FL models, subtracting 4 stages for flow-related computations, only trees of maximum depth up to 6 can be supported. We tailor our models to these constraints.

pForest [40]. Since the source code of pForest is not yet publicly available, we implement a custom version of the mapping scheme. The comparison logic at tree nodes is not trivial to implement and requires an additional stage. This means for trees of maximum depth D, pForest requires $D \times 2$ stages to implement trees in parallel, and an additional stage for deciding the final result of the RF from the individual tree results. With only 12 MAU stages available on the switch, for PL models with simpler logic, we can implement RFs of maximum depth 5. In FL models, $3-5$ MAU stages are required for the flow management and RF voting steps. Thus, only 7 stages are available for tree levels, only allowing trees of maximum depth 3 for pForest, which represents other works that employ the same tree mapping technique like BACKORDERS [48] and SwitchTree [41], both of which are not tested in hardware.

NetBeacon [55] proposes a hybrid solution that can classify both individual packets and flows by deploying multiple tree-based models fully into the user plane. They deploy 3 sets of models. The first is an XGBoost model whose base classifier is a DT, that serves as a flow-size predictor. Its role is to perform a per-packet classification to identify packets that belong to short flows for which no FL features will be computed and stored, or long flows for which memory is allocated and FL features are computed. The second is a PL classifier which classifies all the packets of short flows, packets from flows that suffer a hash collision, and all the packets that arrive before the point where a FL decision can be made. Several such points are identified and used to classify a flow at different phases in its life. We employ the publicly available source code [88] for the implementation of this benchmark. We note that although the paper describes how the proposed model mapping scheme can map RF models with multiple trees, this capability is neither clearly shown in the model design, nor is it illustrated in the released artifacts [88] where only single-tree RFs are used, even for a use case where the RF was indicated to have more trees. As it is not obvious to us how the model can be extended to map multiple trees, we restrained our feature and model selection to RFs with only 1 tree for this benchmark.

2) Categories of comparisons: We perform our evaluation in three steps, comparing `Flowrest` to existing (i) per-packet solutions and (ii) per-flow and hybrid solutions, as well as (iii) testing its peformance under different RF mapping schemes.

(i) `Flowrest` versus packet-level solutions. We compare `Flowrest` against three PL solutions namely Planter [42], Mousika [46] and Soter [52]. For these PL RF models, we train dedicated RFs using only PL features with the Scikit-Learn libraries, and we implement them in hardware by pruning our framework from all its FL operations. This lets us shed light on how `Flowrest` outperforms stateless PL solutions. We stress that this is the very first direct comparison of PL and FL RF models for programmable switches, and the fact that we run it with real-world equipment is a clear added value for the test.

(ii) `Flowrest` versus flow-level solutions. We compare `Flowrest` to a reproduced version of pForest [40], and a recent hybrid packet-flow classifier, NetBeacon [55], to show how `Flowrest` either outperforms or is on par with other stateful classifiers while consuming fewer resources. For each use case, we train the FL model with the same feature set and then implement it using the different approaches.

(iii) `Flowrest` with different RF mappings. We demonstrate the ability of the `Flowrest` framework to accommodate any feasible RF mapping as detailed in Section IV-B by implementing `Flowrest` in 3 additional mappings proposed by Mousika [46], Soter [52], and pForest [40], in addition to the Planter [42] mapping already employed in `Flowrest`. We

train an RF with the same data and features, and then map it into the switch using all 4 mappings embedded into the `Flowrest` framework. We leave out of this tests the mapping proposed by NetBeacon [55] since its ability to handle RFs and not only single DTs is unproven.

D. Metrics

The performance of `Flowrest` and the benchmarks is evaluated using classical metrics: (i) precision, (ii) recall, and (iii) F1-score. These metrics are calculated using fundamental classification measures: true positives (TP), false positives (FP) and false negatives (FN), as follows.

- **Precision** represents the proportion of positive predictions that truly belong to that class, as TP/(TP + FP).
- **Recall** quantifies the number of positive samples that are correctly predicted as positive, as TP/(TP + FN).
- **F1-score**, commonly used to evaluate the performance of models, is defined as the harmonic mean of recall and precision, as 2TP/(2TP + FP + FN).

For every metric, the final value is averaged over all classes in two different ways; (i) a *macro* average which is the simple average of individual class scores; and, (iii) a *weighted* average which assigns weights to individual class scores based on their respective number of samples in the dataset.

As we are interested in classifying flows, we tackle the problem of a fair comparison with scores calculated over packet predictions. First of all, as each flow contains at least one packet, the number of packets is much higher than the number of flows. Besides, the distribution of the flow lengths is also imbalanced over short flows, with up to 50.99% of the flows having less than 2 packets in some of the use cases, and between $47.51\% - 99.87\%$ of flows having less than 50 packets. Hence, we expect the score of the PL solutions be boosted by a good performance in long flows, which have many packets but represent only a small portion of the total number of flows. To better capture the performance of PL solutions to classify flows we normalize the prediction of each packet to the length of the flow it belongs to: that is, we assign to each packet prediction a weight that is inversely proportional to the flow length the packet belongs to, *e.g.*, the weight of a packet in a flow of length l is $w = \frac{1}{l}$. By doing so, the weights of all the predictions of each flow sum to one, and the weights of the predictions of all the packets sum to the total number of flows.

VII. Experimental Results

We start our evaluation by demonstrating the results of the automated model selection process explained in Section IV-A. Figure 8 illustrates all the models derived from the model analysis, where each dot represents a model, with the x-axis corresponding to the average macro and weighted F1-scores, and the y-axis representing the total TCAM usage of the model. The best model (shown in red) that we choose for the in-switch implementation across thousands of options is the one that achieves the best trade-off between accuracy and resource efficiency, according to (1). After selecting the optimal model for each use case, we present the results of our evaluations in terms of memory saving from the simulator output, classification accuracy, switch resource usage, and packet processing latency in the following subsections.

A. Validation of hardware-tailored configuration

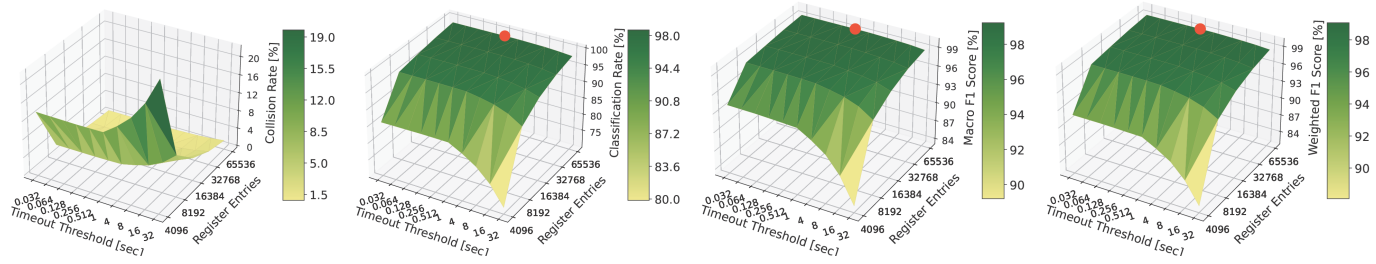
We begin our evaluation with results from the simulator described in Section IV-C, expressed in terms of 4 different metrics; the collision rate, which is the proportion of colliding flows; the classification rate, which is the proportion of flows that get classified; and macro and weighted F1-scores. Figure 7 portrays the simulation results in terms of the four metrics above for the CICIDS and UNIBS datasets in 3D plots, based on two parameters; the timeout threshold in seconds; and the number of register entries used in the flow management tables.

The results show that a careful choice of the timeout threshold and the number of register entries allocated, informed by simulation, can minimize the flow collision rate to nearly 0% in our experiments. In addition, the optimal choice of these parameters also ensures a very high classification rate, higher than 99% in our experiments. Considering the plots of the macro and weighted F1-scores, with the help of the simulator we can save SRAM resources by allocating a smaller number of entries to the flow management tables, while still having a high classification score with only a small drop compared to the scenario with the maximum number of entries. The results of the UNIBS dataset in Figure 7b best illustrate this effect: with only $4,096$ entries and a timeout threshold of 1 second, we can ensure 0% collision rate and 99.4% classification rate, while maintaining macro and weighted F1-scores of more than 96% and 99% respectively. The use of only $4,096$ entries represents a 93.75% saving of SRAM resources over the default $65,536$ entries used. These results demonstrate the usefulness of the simulator in determining the best switch configuration parameters before the actual experiment is conducted in hardware.

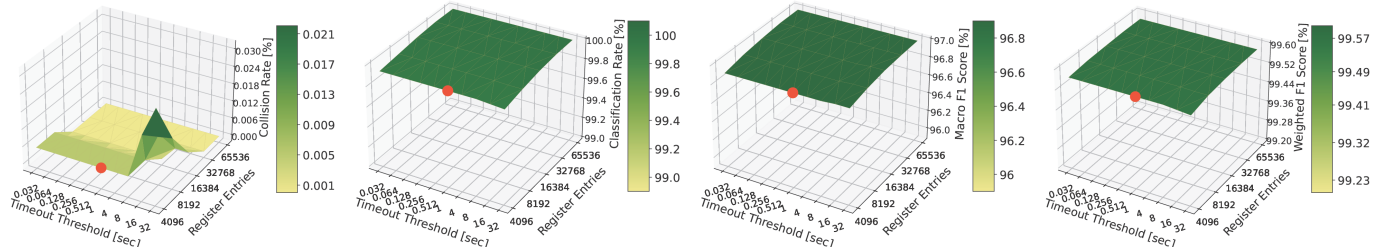
B. Classification accuracy

We evaluate `Flowrest` and the benchmarks via the metrics presented in Section VI-D for the categories of comparison in Section VI-C2. We compare the classification accuracy of `Flowrest` to the per-packet solutions in Table II. Overall, our models outperform their per-packet counterparts in flow classification. Considering the macro precision, `Flowrest` achieves absolute and relative gains over the second-best model for the CICIDS, UNIBS, UNSW and ToN-IoT datasets in the ranges of $1.84\% - 14.22\%$ and $1.94\% - 26.23\%$ respectively, falling short only in the IoT23 dataset where the Planter benchmark slightly outperforms `Flowrest` by 1.06% on absolute terms and 1.12% relatively. The gains of `Flowrest` over the benchmarks are even bigger when considering the weighted precision, going up to 95.11% in the ToN-IoT dataset. This better precision means that `Flowrest` achieves the best quality of positive predictions.

In terms of macro recall, `Flowrest` achieves absolute and relative gains in all use cases, in the ranges of $0.37\% - 14.42\%$ and $0.38\% - 32.09\%$ respectively, while in terms of weighted recall, the absolute and relative gains are in the ranges of

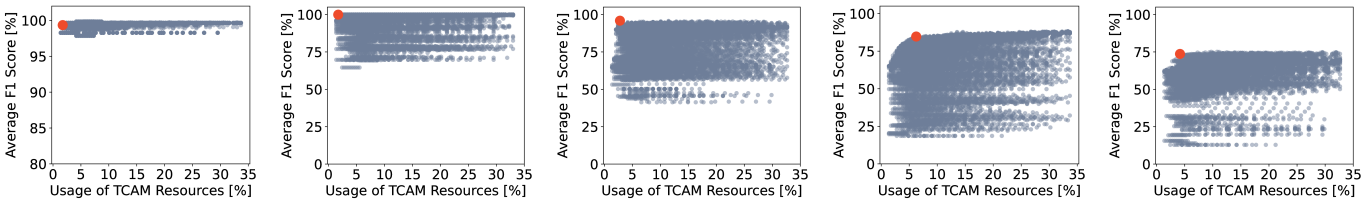


(a) CICIDS



(b) UNIBS

Figure 7: The simulator results of `Flowrest` in terms of the four metrics of the collision rate, classification rate, macro and weighted F1 scores, which are based on the two parameters of the timeout threshold and number of entries used in the flow management tables. The best parameters we pick are highlighted by the red dot.



(a) CICIDS (b) UNIBS (c) IoT23 (d) UNSW (e) ToN-IoT

Figure 8: Scatterplots of the average macro and weighted F1 versus TCAM usage, for all use cases searched by our proposed automated routine. Red dots highlight the selected model for all combinations.

Dataset	Average	Metric	Planter	Mousika	Soter	Flowrest
CICIDS	Macro	Precision	94.448%	87.920%	94.446%	**99.785%**
		Recall	92.900%	78.359%	92.906%	**98.682%**
		F1-Score	93.625%	81.231%	93.628%	**99.231%**
	Weighted	Precision	94.712%	86.668%	94.713%	**99.700%**
		Recall	94.734%	86.009%	94.74%	**98.556%**
		F1-Score	94.688%	85.015%	94.690%	**99.124%**
UNIBS	Macro	Precision	82.561%	86.437%	83.332%	**98.716%**
		Recall	86.183%	86.752%	86.765%	**99.579%**
		F1-Score	83.859%	86.281%	84.360%	**99.111%**
	Weighted	Precision	97.465%	98.802%	97.578%	**99.944%**
		Recall	95.930%	98.247%	95.940 %	**99.862%**
		F1-Score	96.489%	98.500%	96.552%	**99.901%**
IoT23	Macro	Precision	**94.513%**	90.873%	73.912%	93.458%
		Recall	79.191%	87.030%	61.557%	**90.331%**
		F1-Score	84.445%	88.542%	61.943%	**91.621%**
	Weighted	Precision	98.791%	**99.389%**	97.997%	99.315%
		Recall	98.788%	**99.394%**	97.817%	98.577%
		F1-Score	98.693%	**99.386%**	97.662%	98.910%
UNSW	Macro	Precision	54.822%	67.882%	53.608%	**72.839%**
		Recall	57.523%	80.543%	55.677%	**81.760%**
		F1-Score	48.502%	69.103%	47.498%	**72.277%**
	Weighted	Precision	78.597%	90.166%	78.329%	**91.538%**
		Recall	73.906%	88.285%	72.208%	**89.165%**
		F1-Score	73.055%	88.572%	73.084%	**89.733%**
ToN-IoT	Macro	Precision	54.208%	28.728%	51.631%	**68.425%**
		Recall	43.866%	44.916%	41.590%	**59.332%**
		F1-Score	44.711%	30.882%	41.500%	**60.494%**
	Weighted	Precision	44.111%	40.376%	43.671%	**86.067%**
		Recall	44.258%	38.330%	40.947%	**84.928%**
		F1-Score	41.080%	37.463%	39.887%	**85.303%**

Table II: Performance of `Flowrest` compared to the packet-level benchmarks. The best value on each row is in bold.

Dataset	Average	Metric	pForest	NetBeacon	Flowrest
CICIDS	Macro	Precision	99.778%	98.251%	**99.785%**
		Recall	98.690%	**98.918%**	98.682%
		F1-Score	**99.231%**	98.576%	**99.231%**
	Weighted	Precision	99.697%	98.816%	**99.700%**
		Recall	98.556%	**98.793%**	98.556%
		F1-Score	99.123%	98.798%	**99.124%**
UNIBS	Macro	Precision	48.612%	86.905%	**98.716%**
		Recall	53.667%	90.986%	**99.579%**
		F1-Score	50.894%	88.649%	**99.111%**
	Weighted	Precision	94.058%	97.854%	**99.944%**
		Recall	93.966%	97.190%	**99.862%**
		F1-Score	93.934%	97.404%	**99.901%**
IoT23	Macro	Precision	37.294%	85.272%	**93.458%**
		Recall	29.066%	80.766%	**90.331%**
		F1-Score	30.605%	81.528%	**91.621%**
	Weighted	Precision	87.706%	99.022%	**99.315%**
		Recall	88.406%	**99.018%**	98.577%
		F1-Score	86.439%	**98.941%**	98.910%
UNSW	Macro	Precision	14.183%	56.256%	**72.839%**
		Recall	18.412%	66.089%	**81.760%**
		F1-Score	15.200%	53.284%	**72.277%**
	Weighted	Precision	41.582%	81.261%	**91.538%**
		Recall	48.407%	73.394%	**89.165%**
		F1-Score	43.034%	75.470%	**89.733%**
ToN-IoT	Macro	Precision	35.008%	52.521%	**68.425%**
		Recall	32.601%	43.800%	**59.332%**
		F1-Score	31.022%	43.977%	**60.494%**
	Weighted	Precision	71.682%	47.172%	**86.067%**
		Recall	72.252%	45.116%	**84.928%**
		F1-Score	69.019%	40.921%	**85.303%**

Table III: Performance of `Flowrest` and the flow-level benchmarks. The best value on each row is in bold.

Average	Metric	pForest	Soter	Mousika	Planter
Macro	Precision	99.778%	**99.785%**	99.770%	**99.785%**
	Recall	**98.690%**	98.678%	98.652%	98.682%
	F1-Score	**99.231%**	99.228%	99.208%	**99.231%**
Weighted	Precision	99.697%	99.699%	99.683%	**99.700%**
	Recall	**98.556%**	98.550%	98.536%	**98.556%**
	F1-Score	99.123%	99.121%	99.106%	**99.124%**

Table IV: Performance of `Flowrest` when RF models for the CICIDS dataset are encoded with different mapping schemes. The best value on each row is in bold.

Dataset	Model	MACRO			WEIGHTED		
		Precision	Recall	F1-Score	Precision	Recall	F1-Score
CICIDS	Python	99.773%	99.488%	99.629%	99.690%	99.689%	99.689%
	Switch	99.785%	98.682%	99.231%	99.700%	98.556%	99.124%
UNIBS	Python	98.716%	99.759%	99.203%	99.944%	99.939%	99.940%
	Switch	98.716%	99.579%	99.111%	99.944%	99.862%	99.901%
IoT23	Python	92.461%	90.080%	90.972%	99.374%	99.383%	99.351%
	Switch	93.458%	90.331%	91.621%	99.315%	98.577%	98.910%
UNSW	Python	84.051%	82.19%	81.010%	95.32%	94.668%	94.586%
	Switch	72.839%	81.760%	72.277%	91.538%	89.165%	89.733%
ToN-IoT	Python	68.938%	60.988%	62.437%	85.791%	85.341%	85.412%
	Switch	68.425%	59.332%	60.494%	86.067%	84.928%	85.303%

Table V: Performance comparison of `Flowrest`'s offline models against the fine-tuned hardware-aware in-switch models.

$0.88\% - 40.67\%$ and $1.00\% - 91.89\%$ respectively. The only exception here again is in the IoT23 dataset, where the weighted recall of the Mousika benchmark is 0.82% better. These high gains in recall indicate that `Flowrest` returns fewer false negatives than the per-packet benchmarks, which is especially important in security-related use cases.

`Flowrest` outperforms all the per-packet benchmarks in all use cases when the macro F1-score is considered, with absolute gains of up to 15.78% and relative gains of up to 35.30%. When weighted, the gain in F1-score is as high as 44.223% on absolute terms and 107.65% on relative terms. Ultimately, these results validate the superiority of the FL classification enabled by `Flowrest` over existing PL solutions.

The performance of `Flowrest` is then compared to those of other FL solutions in Table III. The results show that in relatively simple datasets like CICIDS where simple models are good enough to solve the classification tasks, all the solutions perform similarly, with `Flowrest` having only a slight gain over pForest [40] and NetBeacon [55] in terms of F1-score. When the use cases become more complex and require larger RF models, the gains of `Flowrest` over the others become more significant in terms of all metrics in the UNIBS, UNSW and ToN-IoT datasets, with absolute gains of up to 18.99% and relative gains of up to 27.30%.

For pForest, the main reason for the drop in performance with increasing use case complexity is the limitation in the maximum depth of trees in the RFs as explained in Section VI-C1. As we could only map models with a maximum depth of 3, all the models predicting more than 8 classes had low scores since there are more classes than tree leaf nodes. In contrast, `Flowrest` models have a maximum depth of up to 20 and thus ensure better performance and scalability. In NetBeacon, a per-packet model is the fall-back model in all cases where a FL classification cannot be made *e.g.*, in the case of a hash collision, when too few packets have arrived, or when the flow is predicted as short. This means that a large number of packets use this model which is generally less accurate and so pulls down the overall score. Instead, `Flowrest` only performs FL classification with a well-tailored flow management scheme to deal with hash collisions, ensuring high accuracy.

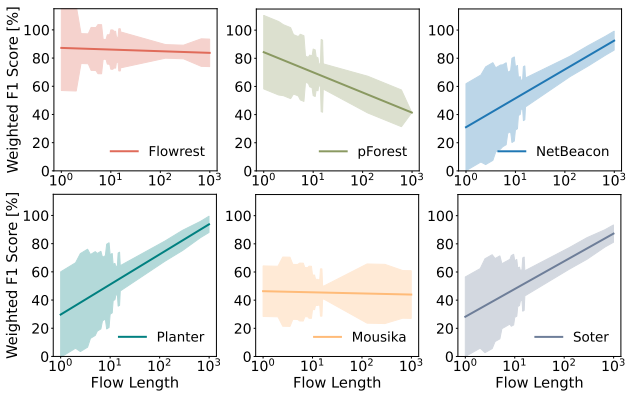


Figure 9: Linear interpolation of the weighted F1 scores on flows of varying lengths. The semi-transparent region surrounding each line illustrates the deviation in scores.

Next, using the CICIDS dataset which requires relatively simple models, we demonstrate in Table IV that any feasible RF mapping can be integrated into the `Flowrest` framework for FL inference. All the models achieve very similar classification results, with slight differences mainly due to the approximations involved with each mapping scheme. This establishes our solution as a viable tool for converting any PL inference system into a FL system.

In Figure 9 we analyze the accuracy of FL and PL solutions in classifying flows with different number of packets (from 1 to $1,000$) in the ToN-IoT dataset. The interpretation of results is facilitated by a linear interpolation (in the logarithmic abscissa scale) of the data points, around which we also plot the deviation observed for flow of different lengths. The plots clearly prove how `Flowrest` yields consistent high inference quality across flows characterized by any length. In fact, `Flowrest` is the only solution attaining this desirable result: the other models favor either low-lasting flows (Planter, Soter, NetBeacon) or short-lived ones (pForest), or perform poorly with flows of any length (Mousika) in the selected use case.

Finally, we investigate the effect of the hardware-aware fine-tuning presented in Section IV-A on model performance. Table V compares the accuracy before and after fine-tuning, showing a minor performance drop in most use cases.

C. Resource usage and latency

We evaluate resource usage in terms of key resources, *i.e.*, TCAM, SRAM, and the average of all other resources. `Flowrest` consumes less TCAM than all the per-packet benchmarks in all datasets except for UNIBS where there is a tie with Mousika, as shown in Figure 10a. Compared to other FL solutions in Figure 10d, `Flowrest` always consumes much less TCAM than NetBeacon, with pForest not consuming any, while suffering from scalability issues. Considering that TCAM is the most expensive and critical resource in programmable switches, the sobriety of our solution is precious. The generally low TCAM consumption attests to the benefit of using Equation 1 to optimize for memory during model selection. With regards to SRAM, `Flowrest` consumes less

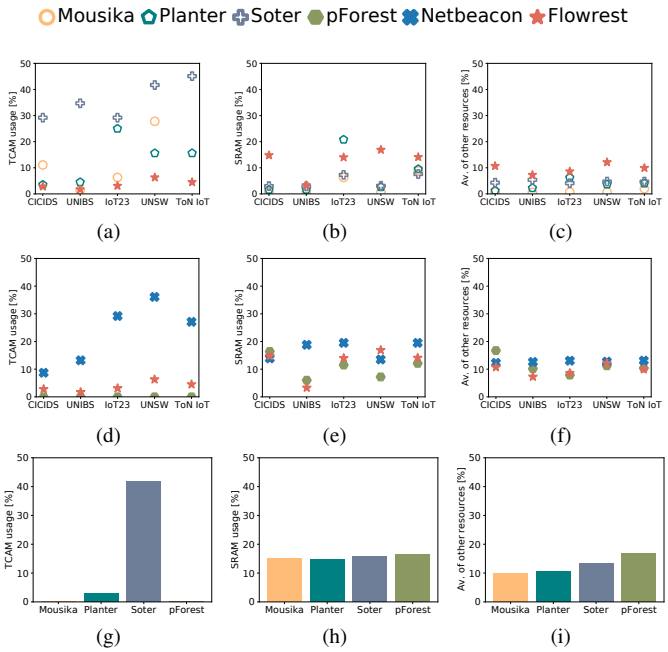


Figure 10: Comparison of resource usage of `Flowrest` vs per-packet benchmarks in terms of (a) TCAM; (b) SRAM; (c) average of the remaining resources; `Flowrest` vs per-flow benchmarks in terms of (d) TCAM; (e) SRAM; (f) average of the remaining resources; and `Flowrest` models for the CICIDS dataset in 4 RF mappings in terms of (g) TCAM; (h) SRAM; (i) average of the remaining resources.

than NetBeacon in 3 out of 5 use cases, but a bit more in the remaining 2. It also consumes a little more than pForest as shown in Figure 10e. In addition, it consumes more SRAM than the per-packet benchmarks due to the need to maintain stateful registers for traffic flows. The only exception is the Planter benchmark in the IoT23 dataset where the model's large voting table with exact matches uses up much SRAM.

Finally, considering the average of all the other resources shown in Figures 10c and 10f, `Flowrest` consumes a bit more resources than the per-packet benchmarks due to the need to maintain state, but always consumes less resources than its closest stateful competitor, NetBeacon. Juxtaposing these consumptions with their classification performances, `Flowrest` achieves by far a better trade-off between memory consumption and classification accuracy, always consuming on average less than 15% of switch resources, leaving room for the switch to implement other functions like forwarding the 40-Gbps background traffic we inject during our experiments.

The ability of `Flowrest` to accommodate any mapping enables to us compare the resource footprint of the mapping schemes from pForest [40], Soter [52], Mousika [46], and Planter [42]. Figures 10g and 10i reveal that while Soter consumes the most TCAM, pForest consumes the most of the other resources, even though it uses no TCAM. Figure 10h shows that irrespective of the mapping scheme used, the SRAM consumption stays relatively the same. This is because SRAM is mainly used to maintain the stateful information for flow classification and since this is dimensioned by the `Flowrest` framework, it is almost constant. The results also

	Mousika	Planter	Soter	pForest	NetBeacon	Flowrest
CICIDS	301.64	318.03	386.89	460.66	419.67	436.07
UNIBS	300.82	336.07	457.38	459.02	436.07	384.43
IoT23	300.82	338.52	440.98	460.66	419.67	364.75
UNSW	303.28	302.46	455.74	459.02	421.31	422.13
ToN-IoT	309.84	336.89	440.98	459.02	419.67	421.31

Table VI: Estimated packet processing latency in *nanoseconds* of the in-switch inference models.

let us conclude that the Mousika mapping generally has the least resource footprint, although its TCAM consumption can at times be very high as under UNSW and ToN-IoT.

All the in-switch models run at line rate. To verify that they achieve sub-microsecond latency, we estimate the delay experienced by a packet that undergoes the inference process when traversing the switch, as described in Section VI-A. As shown in Table VI, all the in-switch models deployed incur sub-microsecond delay. In the case of `Flowrest`, the latency never exceeds 436 ns, just about 136 ns more than the simplest per-packet solution. In addition, when compared to the other FL solutions like pForest and NetBeacon, `Flowrest` typically induces less latency. These results confirm that `Flowrest` can perform line-rate inference in the switch while introducing only a few hundred nanoseconds of delay.

VIII. Conclusions

We proposed `Flowrest`, a practical and general-purpose framework which we empirically demonstrate to be the new state-of-the-art FL in-switch inference system. Our proposed model yields several technical contributions and novel concepts that have general application to user-plane FL, like the synergy of user and control planes for effective line-rate inference, or the design of hardware-aware FL models. Our proposed flow management scheme can accommodate any model mapping and could even serve other non-ML applications which require maintaining FL information. We provide open access to the source code implementing `Flowrest` at `https://github.com/nds-group/Flowrest`.

References

[1] N. Feamster and J. Rexford. Why (and how) networks should run themselves. *CoRR*, abs/1710.11583, 2017.
[2] European Telecommunications Standards Institute (ETSI). Zero-touch network and Service Management (ZSM); Proof of Concept Framework. ETSI GS ZSM 006 V1.2.1, 2022.
[3] J. Xie et al. A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges. *IEEE Commun Surv Tutor*, 21(1), 2019.
[4] C. Zhang et al. Deep learning in mobile and wireless networking: A survey. *IEEE Commun Surv Tutor*, 21(3), 2019.
[5] Y. Zhao et al. A survey of networking applications applying the software defined networking concept based on machine learning. *IEEE Access*, 7, 2019.
[6] A. A. Gebremariam et al. Applications of artificial intelligence and machine learning in the area of SDN and NFV: A survey. In *SSD*, 2019.
[7] K. He et al. Measuring control plane latency in SDN-enabled switches. In *SOSR*, 2015.
[8] H. Chen and T. Benson. The case for making tight control plane latency guarantees in sdn switches. In *SOSR*, pp. 150–156, 2017.
[9] The 5G Infrastructure Association (5G IA). European Vision for the 6G Network Ecosystem. Zenodo, 2021.
[10] D. R. K. Ports and J. Nelson. When should the network be the computer? In *HotOS*, 2019.
[11] Intel. Tofino Programmable Ethernet Switch ASIC. 2016.

[12] Netronome. Netronome Agilio SmartNICs. 2016.
[13] P. Bosshart et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), 2014.
[14] R. Parizotto et al. Offloading machine learning to programmable data planes: A systematic survey. *ACM Comput. Surv.*, 56(1), 2023.
[15] A. Sapio et al. In-network computation is a dumb idea whose time has come. In *HotNets*, NY, USA, 2017. ACM.
[16] M. Hayes et al. Scalable architecture for SDN traffic classification. *IEEE Systems Journal*, 12, 2018.
[17] L. Peng et al. Effectiveness of statistical features for early stage internet traffic identification. *Int. J. Parallel Program.*, 44, 2016.
[18] I. Akbari et al. Traffic classification in an increasingly encrypted web. *Commun. ACM*, 65, 2022.
[19] F. Hu et al. Network traffic classification model based on attention mechanism and spatiotemporal features. *Eurasip J. Inf. Secur.*, 2023.
[20] X. Jin et al. Softcell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*, 2013.
[21] S. C. Madanapalli et al. Reclive: Real-time classification and QoE inference of live video streaming services. In *IEEE/ACM IWQOS*, 2021.
[22] Y. Zeng and T. M. Chen. Classification of traffic flows into QoS classes by unsupervised learning and KNN clustering. *KSII Transactions on Internet and Information Systems*, 3, 2009.
[23] C. Zheng et al. Automating in-network machine learning. *arXiv*, 2022.
[24] G. Li et al. Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In *IEEE ICNP*, pp. 1–12, 2019.
[25] M. Zhang et al. Poseidon: Mitigating volumetric ddos attacks with programmable switches. *NDSS*, 2020.
[26] G. Li et al. IMap: Fast and scalable In-Network scanning with programmable switches. In *NSDI*, pp. 667–681, 2022.
[27] A. Mestres et al. Knowledge-defined networking. *SIGCOMM Comput. Commun. Rev.*, 47(3), 2017.
[28] A. Sapio et al. Scaling distributed machine learning with In-Network aggregation. In *18th NSDI*. USENIX, 2021.
[29] D. Sanvito et al. Can the network be the ai accelerator? In *NetCompute*, NY, USA, 2018. ACM.
[30] D. Barradas et al. Flowlens: Enabling efficient flow classification for ML-based network security applications. In *NDSS*, 2021.
[31] T. Benson. In-network compute: Considered armed and dangerous. In *HotOS*, 2019.
[32] Z. Xiong and N. Zilberman. Do switches dream of machine learning? toward in-network classification. In *HotNets*, 2019.
[33] K. Razavi et al. Distributed DNN serving in the network data plane. In *EuroP4 '22*, NY, USA, 2022. ACM.
[34] G. Siracusano and R. Bifulco. In-network neural networks. *CoRR*, abs/1801.05731, 2018.
[35] M. Courbariaux et al. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Adv Neural Inf Process Syst*, volume 28. Curran Associates, Inc., 2015.
[36] G. Siracusano et al. Re-architecting traffic analysis with neural network interface cards. In *NSDI*, Renton, WA, 2022. USENIX.
[37] N. Corporation. ConnectX SmartNICs - 10/25/40/50/100/200 and 400G Ethernet Network Adapters. 2022.
[38] T. Swamy et al. Taurus: A data plane architecture for per-packet ML. In *ASPLOS*, 2022.
[39] C. Zheng et al. IIsy: Practical in-network classification. *arXiv*, 2022.
[40] C. Busse-Grawitz et al. pForest: In-network inference with random forests. *CoRR*, abs/1909.05680, 2019.
[41] J. Lee and K. P. Singh. Switchtree: in-network computing and traffic analyses with random forests. *Neural Comput. Appl.*, 2020.
[42] C. Zheng and N. Zilberman. Planter: Seeding trees within switches. In *ACM SIGCOMM*, NY, USA, 2021. ACM.
[43] B. M. Xavier et al. Programmable switches for in-networking classification. In *IEEE INFOCOM 2021*.
[44] B. M. Xavier et al. MAP4: A pragmatic framework for in-network machine learning traffic classification. *IEEE Trans. Netw. Service Manag*, 19(4), 2022.
[45] X. Zhang et al. pHeavy: Predicting heavy flows in the programmable data plane. *IEEE Trans. Netw. Service Manag*, 18(4), 2021.
[46] G. Xie et al. Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *INFOCOM*, 2022.
[47] G. Xie et al. Empowering in-network classification in programmable switches by binary decision tree and knowledge distillation. *IEEE/ACM Trans. Netw.*, 2023.
[48] B. Coelho and A. Schaeffer-Filho. BACKORDERS: Using random forests to detect DDoS attacks in programmable data planes. In *EuroP4 '22*, NY, USA, 2022. ACM.
[49] H. Siddique et al. Towards network-accelerated ML-based distributed computer vision systems. In *IEEE ICPADS*, 2021.
[50] A. T. J. Akem et al. Henna: Hierarchical machine learning inference in programmable switches. *NativeNI*, 1, 2022.
[51] B. Bütün et al. Fast detection of cyberattacks on the metaverse through user-plane inference. In *IEEE MetaCom*, 2023.
[52] G. Xie et al. Soter: Deep learning enhanced in-network attack detection based on programmable switches. In *SRDS*, 2022.
[53] K. Friday et al. INC: In-network classification of botnet propagation at line rate. In *Computer Security – ESORICS*, 2022.
[54] K. Friday et al. A learning methodology for line-rate ransomware mitigation with P4 switches. In *NSS*, 2022.
[55] G. Zhou et al. An efficient design of intelligent network data plane. In *USENIX Security*, 2023.
[56] L. Bernaille et al. Traffic classification on the fly. *ACM SIGCOMM Comput. Commun. Rev.*, 36(2), 2006.
[57] M. Crotti et al. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Comput. Commun. Rev.*, 37(1), 2007.
[58] L. Bernaille and R. Teixeira. Early recognition of encrypted applications. In *PAM*. Springer, 2007.
[59] M. Jaber et al. Can we trust the inter-packet time for traffic classification? In *IEEE ICC*, 2011.
[60] G. Lu et al. Comparison and analysis of flow features at the packet level for traffic classification. In *ICCVE*, 2012.
[61] R. Panigrahy and S. Sharma. Reducing TCAM power consumption and increasing throughput. In *HOTI*, 2002.
[62] H. Kim et al. Experience-driven research on programmable networks. *SIGCOMM Comput. Commun. Rev.*, 51(1), 2021.
[63] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.
[64] Wireshark. Tshark. www.wireshark.org/docs/man-pages/tshark.html.
[65] C. Zheng et al. Planter: Rapid prototyping of in-network machine learning inference. *SIGCOMM Comput. Commun. Rev.*, 54(1), 2024.
[66] F. Mola and R. Siciliano. A fast splitting procedure for classification trees. *Statistics and Computing*, 7, 1997.
[67] A. T.-J. Akem et al. Jewel: Resource-efficient joint packet and flow level inference in programmable switches. In *IEEE INFOCOM*, 2024.
[68] S. Liu et al. Leaf: Navigating concept drift in cellular networks. *Proc. of the ACM on Netw.*, 1(CoNEXT2), 2023.
[69] H. Yan et al. Linc: Enabling low-resource in-network classification and incremental model update. In *IEEE ICNP*, 2024.
[70] Q. Zhang et al. Caravan: Practical online learning of In-Network ML models with labeling agents. In *OSDI*, 2024.
[71] F. Pacheco et al. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Commun Surv Tutor*, 21(2), 2019.
[72] B. Hullar et al. Early identification of peer-to-peer traffic. In *ICC*, 2011.
[73] M. Canini et al. Per flow packet sampling for high-speed network monitoring. In *COMSNETS*, pp. 1–10, 2009.
[74] J. Yan et al. Brain-on-Switch: Towards advanced intelligent network data plane via NN-Driven traffic analysis at Line-Speed. In *NSDI*, 2024.
[75] A. Turner and F. Klassen. Tcpreplay, 2013.
[76] P. Emmerich et al. Moongen: A scriptable high-speed packet generator. IMC '15, NY, USA, 2015. ACM.
[77] Intel. P4 Insight. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.
[78] Z. Fang et al. Is out-of-distribution detection learnable? In *NIPS*, 2024.
[79] A. Sivanathan et al. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Trans. Mob. Comput.*, 18(8), 2019.
[80] S. Garcia et al. IoT-23: A labeled dataset with malicious and benign IoT network traffic. Zenodo, 2020.
[81] A. Alsaedi et al. TON_IoT telemetry dataset: A new generation dataset of IoT and IIoT for data-driven intrusion detection systems. *IEEE Access*, 8, 2020.
[82] M. Dusi et al. Detection of encrypted tunnels across network boundaries. *2008 IEEE ICC*, 2008.
[83] A. Este et al. On-line SVM traffic classification. In *IWCMC*, 2011.
[84] I. Sharafaldin et al. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSP 2018*, 2018.
[85] A.T-J. Akem et al. Flowrest. https://github.com/nds-group/Flowrest.
[86] G. Xie et al. Mousika. https://github.com/xgr19/Mousika.
[87] X. Gao et al. Soter. https://github.com/xgr19/Soter/tree/main.
[88] G. Zhou et al. Netbeacon. https://github.com/IDP-code/NetBeacon.

Aristide Tanyi-Jong Akem (Member, IEEE) is a postdoctoral researcher at the University of Oxford. He completed a PhD in Telematics Engineering at IMDEA Networks Institute and Universidad Carlos III de Madrid in 2024. In 2018, he completed an engineering degree at the University of Yaounde I, Cameroon and in 2020, a master's in electrical and computer engineering at Carnegie Mellon University Africa, Rwanda. He has been a visiting researcher at Orange Labs, France, Ranplan Wireless, UK, and the University of Cambridge, UK. Akem's research interests are in programmable networking, machine learning, and related application areas like smart grids, IoT, and next-generation networks.

Beyza Bütün (Student Member, IEEE) is a Ph.D. student in the Networks Data Science Group at IMDEA Networks Institute in Madrid, Spain. She is also a Ph.D. student in the Department of Telematics Engineering at Universidad Carlos III de Madrid, Spain. She earned both her bachelor's and master's degrees in Computer Engineering from Middle East Technical University in Ankara, Turkey, between 2015 and 2022. During her master's, she worked on the optimal design of wireless data center networks. Beyza's current research interests are machine learning-based inference, distributed in-band network intelligence, and energy consumption modeling and optimization in the data plane.

Michele Gucciardo (Member, IEEE) is a Research Engineer at NEC Laboratories Europe. He received the B.Sc. degree from Politecnico di Milano, Italy (2008), and the M.Sc. degree from the University of Palermo, Italy (2013), all in telecommunications engineering. He later received the Ph.D. degree in information and communication technologies from the University of Palermo (2020). He has also been a visiting researcher at the TIM Labs, Italy (2018) and a postdoctoral researcher at the IMDEA Networks Institute, Spain (2021-2024). His research interests have been focused on IoT, software defined and programmable networks, data plane inference, machine learning and generative AI for beyond 5G networks.

Marco Fiore (Senior Member, IEEE) is a Research Professor at IMDEA Networks Institute, where he leads the Networks Data Science group, and co-founder and CTO at Net AI, a UK-based network intelligence company. He received MSc degrees from University of Illinois at Chicago and Politecnico of Torino, a PhD degree from Politecnico di Torino, and a Habilitation à Diriger des Recherches from Université de Lyon. Marco has held tenured positions at Institut National des Sciences Appliquées de Lyon and National Research Council of Italy, and has been a visiting researcher at Rice University, Universitat Politècnica de Catalunya, and University College London. Marco's research is at the interface of mobile networks and data science, and has received multi-million Euro funding from the European Commission and national agencies in Spain, France and Italy, as well as a number of recognitions that include two best paper awards at IEEE INFOCOM. Marco is a former Marie Curie fellow and Royal Society visiting research fellow, and a Senior Member of IEEE and ACM. His research is the interface of computer networks, data science and machine learning.