This is the author's accepted version of the article. The final version published by IEEE is A.T-J. Akem, M. Gucciardo and M. Fiore, "Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests," IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, 2023, pp. TBD, doi: TBD.

Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests

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

***Abstract*—User-plane machine learning facilitates low-latency, high-throughput inference at line rate. Yet, user planes are highly constrained environments, and restrictions are especially marked in programmable switches with limited memory and minimum support for mathematical operations or data types. Thus, current solutions for in-switch inference that are compatible with production-level hardware lack support for complex features or suffer from limited scalability, and hit performance barriers in complex tasks involving large decision spaces. To address this limitation, we present `Flowrest`, a first complete Random Forest (RF) model implementation that operates at the level of individual flows in commercial switches. Our solution builds on (i) an original framework to embed flow-level machine learning models into programmable switch ASICs, and (ii) novel guidelines for tailoring RF models to operations in programmable switches already at the design stage. We implement `Flowrest` as an open-source software using the P4 language, and assess its performance in an experimental platform based on Intel Tofino switches. Tests with tasks of unprecedented complexity show how our model can improve accuracy by up to 39% over previous approaches to implement RF models in real-world equipment.**

I. Introduction

Over the past decade, the growing flexibility and complexity of network architectures has been calling for increasingly automated network operations that reduce or even entirely replace conventional human-in-the-loop approaches. Concepts such as self-driving networking [1] or zero-touch network and service management (ZSM) [2] envision networked systems where controllers and orchestrators collect measurements and analyze them to feed policies and algorithms that can take effective management decisions in real time or even proactively.

In this context, the success of Software Defined Networking (SDN) has paved the road for the deployment of machine learning models in control planes, where they automate a wide range of management functions. Examples include traffic classification, quality of service (QoS) prediction, routing optimization or security enhancement, among others, as comprehensively reviewed in several recent surveys [3]–[6]. Although intelligent functionalities such as those listed above represent a substantial contribution to automating network operations, control-plane solutions show inherent limitations for real-time decision-making at high speed. Indeed, the need for communication from and to the user plane introduces a structural delay that prevents inference at line rate –a key requirement for self-driving networking [1], and a critical technical achievement towards meeting next-generation mobile networks sub-millisecond end-to-end latency specifications [7].

Inference at line rate needs pure user-plane implementations of machine learning models, alongside in-network computation principles [8]. The advent of programmable user planes triggered activities in that direction: the availability of commercial protocol-independent and programmable ASICs [9] and Network Processing Units (NPUs) [10], alongside dedicated programming languages such as P4 [11], has led to proposals to embed different models directly into switches or Smart Network Interface Cards (SmartNICs).

The endeavour is especially challenging in the case of programmable switches, due to their severe limitations in terms of memory, support for mathematical operations and number of allowed operations per packet [8], [12]. Prior studies have shown the promises of in-switch inference based on Decision Tree (DT) and Random Forest (RF) models. Yet, owing to the constraints above and as thoroughly discussed in Section II, state-of-the-art approaches have shortcomings in terms of: (i) design, *e.g.*, by relying only on basic features extracted independently for each packet, and lacking flow statistics that are instrumental to an effective inference such as inter-arrival times or flow-level counts; (ii) scalability, *e.g.*, of the machine learning model size, and of the complexity of tasks it can address effectively; or, (iii) practical viability, *e.g.*, due to incompatibilities with real hardware. As a result, to date, *no practical solution is available that can run Random Forest models at flow level in real-world programmable switches*.

In this work, we propose a first complete design that closes the aforementioned gap. Our solution, `Flowrest`, allows integrating large RF models in production-grade programmable hardware, so as to perform challenging inference tasks on individual traffic flows at line-rate. The development of `Flowrest` sets forth the following main contributions.

- We present a pragmatic framework to embed generic flow-level machine learning models into commercial programmable switch ASICs. To this end, we propose solutions to (i) run the legacy forwarding logic of the switch and the line-rate inference process in a synergic way, and (ii) effectively manage and exploit the stateful per-flow information that is beneficial to complex tasks.
- We draw original guidelines for the generation of RF models that are natively tailored to the requirements of commercial programmable switches. This ensures that the strict constraints of network hardware are considered by design, during the phases of (i) feature engineering and (ii) machine learning model hyper-parameter tuning.

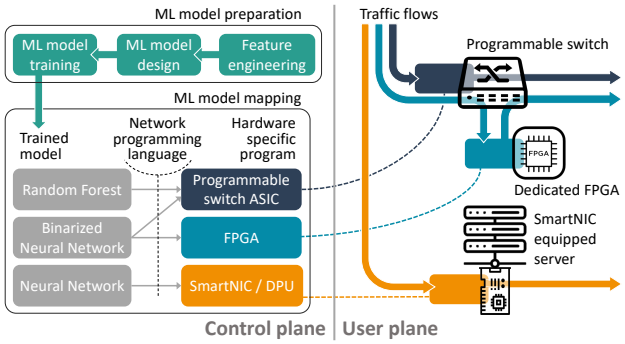


Figure 1: Summary of the approaches for line-rate inference.

- Our framework can accommodate any mapping of the RF model onto Protocol Independent Switch Architecture (PISA) pipelines. We then realize a first complete solution for in-switch flow-level RF-based inference, which we name `Flowrest`, by reproducing and integrating in the framework a state-of-the-art RF mapping.
- We implement `Flowrest` into a real-world Intel Tofino switch using the P4 language, along with a benchmark mimicking current packet-level approaches in the literature. Our implementation is open source, which reduces the significant barriers that exist today in the access to hardware-level code of solutions for in-switch inference.
- We run experiments in a real-world testbed, revealing how `Flowrest` yields increasing accuracy gains over a packet-level model as the complexity of the inference task grows. In a device identification problem with decision space of 26 categories, `Flowrest` improves accuracy by 39%, without increasing the use of key switch resources such as the Ternary Content Addressable Memory (TCAM) or Packet Header Vector (PHV).

Overall, these contributions let us advance the state of the art in in-switch inference, and make steps towards a deeper integration of machine learning into network equipment.

II. Line-rate inference: a primer

We provide a brief but complete review of existing solutions for line-rate inference, looking at the whole user plane, in Section II-A, and then focusing on in-switch approaches, in Section II-B. As our study focuses on pure user-plane implementations, we do not consider hybrid strategies that split the inference process across planes [12]–[15].

A. Machine learning in the user plane

Figure 1 offers a unifying view of the workflows adopted for line-rate inference. Given the limitations of programmable network hardware, all existing approaches in scope of our study assume that the whole model preparation, including computationally expensive phases such as model hyper-parameter optimization and training, is performed offline in the control plane, typically using dedicated Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). Trained models are then encoded for operation in the target user-plane equipment via a network programming language. Where proposals vary are in terms of (i) the family of machine learning models considered for user-plane implementation, and (ii) the nature of the target programmable hardware. We next review prior works based on the hardware they target, while also discussing the specific models they implement.

Programmable switch ASICs. The majority of the effort has focused to date on in-switch inference. In this case, as shown in Figure 1, the machine learning model is fully deployed in a single programmable switch, by implementing it into the switch ASIC: this allows processing packets at line-rate using solely the resources of the switch. Here, Decision Tree (DT) and Random Forest (RF, realized by running multiple DTs in parallel) are common choices of model, due to their relatively low complexity and intuitive mapping to the ASIC pipeline. As this is the strategy we also adopt for our solution, we provide an in-depth discussion of current in-switch implementations of DTs and RFs in Section II-B.

Other machine learning models have been also tested for in-switch operation. The list includes Support Vector Machines (SVM), Naive Bayes, K-Means, XGBoost, or Isolation Forest; however, DTs and RFs were ultimately found to offer higher scalability and better performance [16], [17]. The same holds for attempts targeting more complex models, such as Neural Networks (NN) [17], [18], possibly with a Binarized flavor (BNN) [19] that relies on $+1/-1$ weights and sign activations [20]. Even simplified versions of neural networks are onerous to deploy into commodity programmable switches: a very basic BNN with two layers of 64 and 32 neurons already exhausts the resources of an Intel Tofino ASIC [19], while yielding poor inference results due to its exceeding simplicity.

SmartNICs. The fundamental limits of programmable switches in supporting neural networks have fostered the exploration of alternative user plane hardware where to implement such models. N3IC [21] is a very recent work that presents a comprehensive approach for integration of BNNs on SmartNICs. Figure 1 highlights how N3IC maps the machine learning model to a SmartNIC hardware located in a server.

The SmartNIC environment alleviates the constraints of programmable switches, and N3IC can implement a 3-layer BNN that operates at line-rate while consuming a relatively small fraction of the available resources. Yet, SmartNICs are deployed at network appliances (*e.g.*, traffic classifiers, load balancers, or security middleboxes) that physically reside in a dedicated host within the network datacenter. Approaches like N3IC can thus grant line-rate inference at specific locations of the network only, and not at any point of the transport domain as it can potentially happen with in-switch solutions. We also note that a single SmartNIC is priced similarly to a production-level programmable switch, while supporting a much smaller number of ports (1–4 versus 16–64), and is thus substantially less cost-efficient on a per-port basis [22].

FPGA-enhanced switches. An even more radical strategy is that of adding dedicated FPGAs to switches or SmartNICs, and exploit them to implement complex NN models. The recently proposed Taurus [23] framework employs a custom accelerator based on a pipelined Single Instruction Multiple Data (SIMD)

parallelism to implement NNs via MapReduce operations. Taurus-enhanced switches offload to the external hardware the per-packet inference process, enabling the deployment of powerful machine learning models in user planes.

The limit of the approach taken by Taurus obviously lies in substantial added costs and technical complexity. Its adoption at scale would require revisiting the user plane design, and deploy significant custom hardware next to already expensive programmable switches and SmartNICs.

B. In-switch tree-based inference

We focus our attention on in-switch inference with DT and RF models. The rationale is twofold and is based on the overview above: first, running machine learning models on standalone programmable switch ASICs paves the way to the most pervasive user-plane deployment possible at no additional hardware cost; second, tree-based models are the most performing solutions in such very challenging environments.

As anticipated, several studies have investigated in-switch tree-based inference. Table I summarizes all such previous works, based on three classes of features: design strategy, scalability to complex models and decision spaces, and suitability to practical environments. Next, we detail prior solutions and juxtapose them to ours along the dimensions above.

Design. In terms of design, the key dichotomy is between inference at packet versus flow level. In packet-level designs, the models are fed with features gathered independently from the headers of each packet. This approach is simpler to realize, as the feature extraction only builds on baseline header parsing functionalities of programmable switches. However, it also has significant drawbacks. First, packet-level models do not have access to features computed over the whole flow, such as inter-arrival times or flow-level counts; these features have repeatedly proven paramount to accurately solve complex network tasks [32]–[36]. Second, packet-level models are stateless and need to be run on each and every packet transiting through the switch; this entails significant additional costs in terms of processing, for increased table lookups, and energy, due to higher access to power-hungry TCAM [37].

Flow-level inference relies instead on statistical features that can be collected over multiple packets of a same flow. This has the potential to overcome the limitations above by (i) enabling access to flow-level statistics, and (ii) producing decisions that apply to all packets of a flow while processing just a few of them. However, it also prompts new design challenges. As a result, despite its advantages, only a minority of the solutions in the literature, *e.g.*, pForest [25], NERDS [28] and pHeavy [29], support flow-level operation. `Flowrest` advances the literature above by proposing a pragmatic framework to embed flow-level machine learning models into commercial programmable switch ASICs, in Section III.

Moreover, a unique aspect of our solution is that it proposes and takes advantage of guidelines for the design of RFs that ensure the compatibility of the trained models with off-the-shelf programmable equipment. pForest [25] and SwitchTree [26] have included feature selection as part of their operation, which is however a legacy model preparation step, completely agnostic of in-switch requirements. pForest [25] has proposed a feature compression technique to save switch resource, yet the approach is not viable in practice[1]. Unlike such previous proposals, `Flowrest` for the first time tailors feature engineering and hyper-parameter tuning to the underlying user-plane operation, as detailed in Section III-E.

Scalability. We look at scalability from two perspectives: (i) the size of the tree-based model supported; and, (ii) the largest inference task demonstrated with experiments.

In terms of complexity, most solutions allow multi-tree RF designs. Yet, the most recent approaches, NERDS [28], pHeavy [29] and Mousika [30] are limited to single-tree models, *i.e.*, DTs. This is an obvious shortcoming, as RFs generalize DTs and are widely recognized to have superior learning capability, especially when the task difficulty grows.

Existing designs employ different strategies to map DTs or RFs into the Protocol Independent Switch Architecture (PISA) adopted by modern programmable user planes. Most strategies introduce structural limits to the maximum tree depth. For instance, the mapping of pForest [25], later reused by SwitchTree [26], associates each tree level to one Match-Action Unit (MAU) stage of the PISA pipeline; hence, the tree depth is bounded by the small number[2] of MAUs in commercial switch ASICs. Also, NERDS [28] and pHeavy [29] employ mappings where conditional statements and/or different trees are associated to individual MAU stages and must be executed in series. This sequential operation across limited MAUs creates an inherent ceiling to the model complexity.

Other works employ encodings of the tree that decouple the levels from MAUs, removing the systemic limitations above. In particular, the mapping strategy first proposed by Ilsy [16] and later extended by Planter [27] is the state of the art, since it can embed in real-world hardware multiple trees whose depth is only delimited by the switch memory. Thus, while `Flowrest` can accommodate any technique to map the RF model into PISA pipelines, it presently relies on a custom implementation of this mapping technique, expounded in Section III-D, to ensure scalable multi-tree support.

The heterogeneous complexity of the machine learning models realized by each solution proposed in the literature translates into demonstrated use cases of very varied difficulty. In Table I, we report the maximum number of target traffic classes that are to be inferred in the performance evaluation of each study; while we acknowledge that the cardinality of the classification problem only captures part of the complexity of an inference task, we believe that is it a simple, reasonable quantitative metric to tell apart naive and harder case studies.

[1]The bit-level compression of features proposed in pForest [25] allows saving a few bits in the representation of each feature, yet commercial programmable switch ASICs only support byte-level memory allocation, making the vast majority of the bit-level optimizations unprofitable in hardware.

[2]Intel Tofino switches have 12 MAU stages; for flow-level inference, some stages must be dedicated to calculating flow identifiers, maintaining register indices, managing stateful features, and implementing tree leaves, curbing the maximum attainable tree depth. For instance, pForest [25] claims that only rather shallow trees of depth 4 can be been implemented in such hardware.

Solution	Design		Scalability			Practicality			
	Hardware-tailored model preparation	Flow-level support	Forest support	Unrestricted tree depth	Largest use case	General purpose	Hardware implementation	Resource usage analysis	Open code
Ilsy [16], [24]			✓	✓	2	✓	✓		
pForest [25]		✓	✓		8	✓			
SwitchTree [26]		✓	✓		2	✓			✓
Planter [17], [27]			✓	✓	3	✓	✓		
NERDS [28]		✓			7	✓			✓
pHeavy [29]		✓			2		✓		
Mousika [30]				✓	6	✓	✓	✓	✓
BACKORDERS [31]		✓	✓		2				
Flowrest	✓	✓	✓	✓	**26**	✓	✓	✓	✓

Table I: Comparative summary of prior solutions for in-switch inference and Flowrest. Columns refer to (i) adoption of a machine learning modelling approach that is 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) largest use case demonstrated in number of classes, (vi) applicability to general inference problems opposed to solving a dedicated task only, (vii) implementation and experimental evaluation with a real-world hardware platform, ($viii$) complete analysis of switch resource consumption based on memory types, (ix) availability of open-source code.

It is interesting to observe that multiple works have only looked at binary classification problems, and that the maximum number of traffic categories considered across the whole literature is 8 in emulated environments and just 6 in real-world testbeds. We show in Section IV how Flowrest can tackle much more challenging tasks involving up to 26 classes.

Practicality. In-switch inference solutions can be developed for software or hardware targets, using a same network programming language like P4. In the former case, the solution can be only evaluated in emulation, *e.g.*, by running the popular *bmv2* target within a Mininet environment. Although useful for initial development and debugging, software implementations are very distant from production-level hardware targets, such as Intel Tofino ASICs. Not only throughput and latency are completely different, but emulation hides many hard constraints of real-world equipment: as a result, solutions that are only tested in software are typically not compatible with actual programmable switches, and porting them requires substantial re-design and results in performance losses [38].

Therefore, the main trait defining the practical viability of a model is whether it is implemented in real-world hardware. As per Table I, this holds for the most recent versions of Ilsy [24] and Planter [17], which however only support packet-level inference. Other fresh works, pHeavy [29] and Mousika [30], also present implementations in hardware, yet they only accommodate single-tree models. Also, code is publicly available only for the last approach among the four mentioned above.

With Flowrest, we provide a first solution for in-switch inference that can operate on production-level equipment, and supports both flow-level features and multi-tree models. This sets a new standard for the state of the art, which we open source to foster further research about in-switch inference.

Finally, we comment on two further aspects in Table I. First, while the majority of the proposed models are general-purpose and can be applied to varied inference tasks, pHeavy [29] is dedicated to a specific goal, *i.e.*, the binary identification of heavy flows. Second, with the exception of the packet-level Mousika [30] model, no previous study detailed the requirements in terms of switch resources. We instead propose a general-purpose model, and provide a first look at resource usage of a flow-level model in production-level equipment.

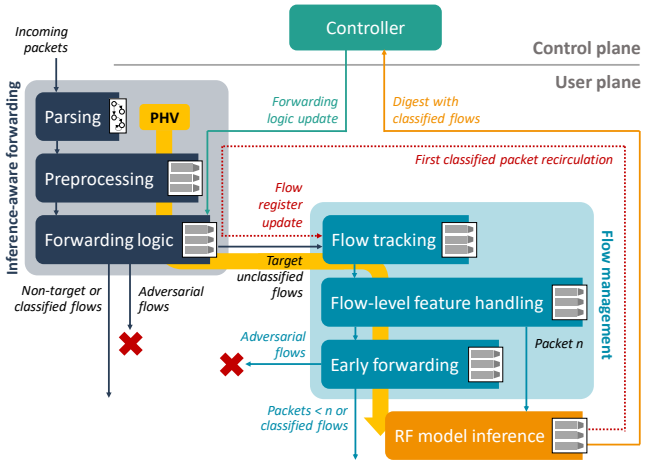


Figure 2: Overview of the system proposed for Flowrest.

III. Practical Flow-level In-switch Inference

Our Flowrest solution sits in a complete system for user-plane flow-level inference, portrayed in Figure 2. The system mainly rests in the programmable switch, but also encompasses control plane functions via, *e.g.*, an ONOS controller. Next, we detail our proposal, and highlight its original aspects.

A. Parsing

Incoming packets at the switch are processed by a PISA pipeline composed by a parser followed by a sequence of Match-Action Units (MAU). The parser extracts metadata from the headers of each packet. While the metadata is only functional to forwarding decisions in standard switch operations, we program the parser so as to collect packet-level information that is also relevant to the target inference task. This is stored into the Packet Header Vector (PHV), *i.e.*, a set of containers that carry raw header fields of interest (*e.g.*, the payload size) and gathered metadata (*e.g.*, the packet arrival timestamp) along the whole MAU pipeline. We remark that this is analogous to the packet data handling of pForest [25].

B. Inference-aware forwarding

Packets then enter the MAU stages implementing the legacy behavior of the switch, *i.e.*, the dedicated pre-processing of

the metadata required by advanced forwarding rules, and the actual forwarding logic. However, in `Flowrest`, *we integrate inference-related configurations into the forwarding logic itself*, and let the controller specify the following elements.

- The target for inference, *i.e.*, the part of traffic concerned with the inference task. For example, the network manager may target a specific range of source IP addresses suspected to host malicious activities, or a set of protocols known to support the traffic to be classified. The selection is implemented as part of the forwarding logic, and flags packets that shall go through in-switch inference.
- The status of packets that are among the inference targets, *i.e.*, whether their flow has been already classified or not. If yes, the controller configures the switch to deactivate the flag above, as inference is not needed anymore.
- The forwarding decision for already classified flows. The controller configures the forwarding logic to handle classified flows based on the outcome of the RF model.

The forwarding logic thus directly acts upon flows that are not targets for inference, or for which RF results are available. As exemplified in Figure 2, regular traffic is directed to the egress ports based on legacy rules, while, *e.g.*, flows identified as adversarial by the inference process are dropped right away.

Clearly, this requires that the controller is aware of the inference results and uses them to configure the forwarding pipeline. To this end, once a flow has been classified by the RF model as later explained in Section III-D, the switch is programmed to inform the controller via a *digest*, *i.e.*, a small and flexible message used exclusively for communications to the control plane. The digest contains the unique flow identifier introduced in Section III-C and the associated inference result, *i.e.*, its class. The controller can then leverage the received information to update the forwarding logic, as per Figure 2.

We acknowledge that the closed loop with the control plane breaks the concept of a pure user-plane inference. Yet, the fact that the forwarding logic is reconfigured by the controller with a significant delay of seconds or higher is not a problem in our framework. Indeed, until the instant when the forwarding logic is updated with the RF model result for a given flow, the decisions for the associated packets are taken fully in the switch, via the flow management routine that we will present in Section III-C. Ultimately, the system abides by the specifications of pure in-network computation: it ensures that inference happens at all times at line rate and with very low latency, be it in the flow management MAUs at first or in the forwarding MAUs upon a closed loop with the controller.

As a result, the architectural solution adopted by `Flowrest` in Figure 2 lets the forwarding and inference stages operate in synergy. The advantages are significant. Processing is faster since packets that have already been (or do not need to be) classified only go through the regular forwarding logic. Also, the model scales better, as it leverages the forwarding tables to offload already classified flows from the flow management registers that, as seen in Section III-C, are limited in size.

We finally remark that the system above is not feasible with packet-level inference where flows are not differentiated. And it is novel with respect to all previous solutions for flow-level inference, which are bounded to in-switch operation, and do not set forth a complete system where the RF model benefits form a deep integration with regular forwarding pipelines.

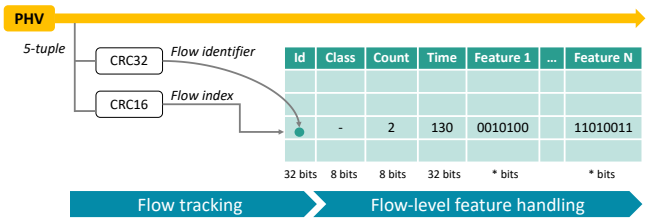


Figure 3: Flow management table and its associated stages.

C. Flow management

Packets that the forwarding logic tags as inference targets, and for which a class is not yet available in the forwarding pipeline, are moved to flow management. This stage deals with the computation, storage and usage of flow-level information. It is decomposed into three phases, detailed next, which rely on a same *flow management table*, represented in Figure 3. The table stores all data relevant to each flow currently under inference, and is implemented as a set of registers in the Static Random Access Memory (SRAM), and contains (i) the identifier of the flow, (ii) its class if already determined, and (iii) all of its stateful features.

Flow tracking. This component is in charge of identifying incoming flows, and monitoring their status in the flow management table. First, a *flow identifier* is generated for the packet, via a CRC32 hash checksum of the 5-tuple composed of IP source and destination addresses, source and destination ports, and transport protocol identifier. We also compute a *flow index* as a CRC16 hash of the same 5-tuple, used to locate the entry for the current flow in the flow management table.

As shown in Figure 3, when accessing the flow management table with the flow index, we first compare the flow identifier with that stored in the table. If the two match or the entry is empty, we proceed to the following phase. A mismatch, denoting a hash collision, prevents storing stateful information and thus no inference is performed on the flow. Clearly, flow index collisions are highly undesirable events that impact the overall performance of the in-switch inference process. Unfortunately, the limited size of registers and the large number of flows traversing a switch may make such events frequent[3]. Previous proposals to cascade hashes in emulated environments [25] are very expensive to implement in hardware as they require a dedicated MAU stage, and we opt not to employ them. Instead, we find that the integration with the forwarding logic outlined in Section III-B naturally mitigates collisions: by offloading the inference-based decision to the regular forwarding pipeline for a given flow, we can remove the corresponding flow entry from the flow management table. This limits the number of flows concurrently present in the table, and dramatically reduces collisions[4].

[3]In our use cases, more than 10% of flows collide without mitigation.

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

To remove flows from the management table, we employ a timeout, which is much simpler than triggering the deletion of the entry right upon inference. Every time a collision occurs, we check the time elapsed since the arrival of the last packet of the flow already in the table: if it is above a threshold[5], we purge the data from the entry and accommodate the new flow.

Flow-level feature handling. Once packets pass the flow tracking phase, we compute, update and store all data in the flow management table. As shown in Figure 3, each entry contains the class of the flow if available, the count of recorded packets in the flow, and a timestamp of the arrival of the last packet used for the entry timeout as described above. In addition, it stores all the stateful features needed by the flow-level RF model. The exact list of features depends on the specific RF model, and, in our tests, is always a subset of the complete flow-level features listed in Section III-E.

Several important remarks are in order about this phase.

- From a technical standpoint, all the stateful variables in the flow management table must be read and updated at the same time, which is a non-trivial operation in hardware. We achieve it via the Tofino Native Architecture (TNA) RegisterAction extern function that exploits Stateful Arithmetic Logic Unit (S-ALU) associated to the registers for mathematical and logical operation.
- The counter is used to implement an *early-flow detection* approach [39]. Namely, `Flowrest` performs the inference upon reception of the first few packets of each flow as also done by pForest [25]. Only a single n-th packet in the sequence of the flow[6] is processed by the RF model; this is a further advantage over packet-level inference that must process all packets in each flow.
- The class indicates the result of the RF model for the flow. To store this information, we exploit the re-circulation port as shown in Figure 2. After a packet has completed the inference pipeline, we attach to it a custom header with the output class information and re-circulate it. Since the packet crosses the forwarding stages exactly as during its first passage, it is handed again to flow management. There, a dedicated logic in the flow-level feature handling phase uses the custom header to update the class in the corresponding flow management table, strips such a header, stores the class in the PHV and forwards the packet to the early forwarding phase.

Early forwarding. By coupling an early-flow detection and class re-circulation, `Flowrest` can immediately react to the outcome of the inference process. Once a packet reaches the flow management stage and is found to belong to an already classified flow, it is directed to the early forwarding stage. Here, we implement a dedicated logic for already classified flows, which takes forwarding decisions based on the class reported in the flow management table. Figure 2 shows how, *e.g.*, all packets beyond the n-th of a flow classified as malicious are directly dropped at this stage.

[5]A timeout of 60 seconds proved to work well in all use cases we analyzed.

[6]In our experiments, we use early-flow detection with the third packet, n=3.

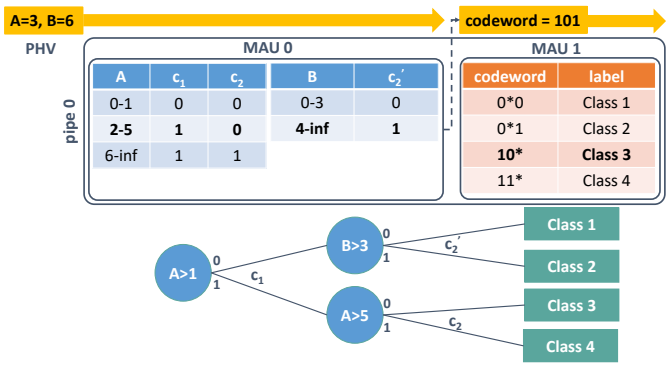


Figure 4: Overview of the Planter RF model mapping to MAUs adopted by `Flowrest`. Each feature employed by the RF is mapped to one MAU, and different trees of the forest share the same match-action tables (except for the last one). Moreover, the compiler may optimize memory usage by storing multiple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

As already mentioned in Section III-B, the early forwarding is run in the flow management for each recently categorized flow, until the controller is informed of the flow class and the main forwarding logic is updated to reflect that.

Finally, we also note that the early forwarding phase is also configured to handle packets *before* the n-th in each flow. While they contribute to flow-level statistics in the feature handling phase, these packets come too early in the flow to be classified; passing them on to the early forwarding ensures that they are processed, *e.g.*, based on standard rules.

D. RF model mapping and inference

When the flow management processes the n-th packet of a target not-classified flow, the flow-level features are stored in the PHV along the packet-level ones, and run across an RF model encoded in the remaining MAU pipeline, as shown in Figure 2. The framework developed for `Flowrest` can accommodate any hardware-feasible model or mapping strategy without changes to the workflow stages presented before. To the best of our knowledge, ours is the first framework for flow-level in-switch inference that allows for easy integration of different model mapping strategies.

In order to build a complete solution for in-switch flow-level RF-based inference, we reproduce and integrate in the framework a custom version[7] of the mapping recently proposed by Planter [27]. The rationale for this choice is that, as discussed in Section II, this is a state-of-the-art mapping strategy ensuring scalable multi-tree support. Below, we summarize the functioning of the Planter RF mapping.

The characteristic element of the mapping is that individual match-action operations are mapped onto single features, so that each feature is processed in exactly one MAU even if used across many levels of different trees. This approach implies that all decisions based on a same feature and performed at all nodes of all trees in the RF must be handled at once. This is illustrated by the toy example in Figure 4; the diagram refers

[7]The code of Planter is not openly available at the time of writing.

to a single tree, but it can be easily generalized to the forest case. First, value ranges are computed for each feature so as to capture all thresholds the feature may encounter in the RF, and are stored into the match part of a MAU. For instance, in Figure 4, the feature A is compared to thresholds 1 and 5 at nodes C1 and C2, respectively: hence, the match-action table associated to A reports three value ranges, *i.e.*, less than or equal to 1, between 2 and 5, or 6 and above.

Then, for each node using the feature, a result true (1) or false (0) is associated to each value range above, via the action stage. For instance, 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 $A \in [6, \infty)$. The second decision C2 is instead verified at 1 only in the last case, $A \in [6, \infty)$. Through the match-action operation, a packet with A=3 is assigned a code 10 for feature A, which is stored into PHV metadata.

By repeating the process for each feature and concatenating the action results in the PHV, one builds a full *path code* for the packet. In the last MAU, each tree in the forest has a dedicated match-action sequence, where the path code is matched against predefined sequences that describe all possible paths from the root to the leaves of the tree. This allows matching the packet with a specific path, hence its associated label, which is stored in the PHV and re-circulated as presented in Section III-C.

An important remark is that not all possible path codes are feasible in each tree, since each bit refers to one node, and a path only traverses a subset of nodes. Wildcards are thus employed in the last match stage to represent bits (*i.e.*, nodes) that are not along the path, and whose value is thus irrelevant to the current classification decision. This determines that in the final MAU the mapping heavily relies on Ternary Content Addressable Memory (TCAM), required for non-exact matches with feature value ranges and wildcards.

E. Switch-tailored model design

A distinguishing aspect in the operation of `Flowrest` is the fact that *we account for the hardware constraints already at the stage of preparing the machine learning model*. This facet of `Flowrest` is orthogonal to the view in Figure 2, as it covers the stages of model preparation shown Figure 1, which are implemented in the control plane and run offline.

Our hardware-tailored modeling process stems from the way flow management and RF inference mapping are executed, and ensures that the trained RFs are compatible with the programmable switch *by design*. Like most previous works, we rely on the popular Scikit-Learn libraries [40] for the model preparation stage, but we tailor the feature engineering and the selection of hyperparameters to accommodate implementation into real-world equipment. Our complete machine learning pipeline is organized along the following steps.

Feature extraction. Both packet-level and flow-level features are computed in the training data, *e.g.*, using Tshark [41] to extract packet header information from historical pcap traces. While any set of features of interest can be gathered at this stage, for all the experiments in this paper we used as stateless packet-level features the TCP/UDP source and destination ports, packet length, and TCP flags (ACK, SYN, PUSH, ECE, RESET, FIN). For stateful flow-level features, we employ statistics on the packet length (total, minimum, maximum and mean), inter-arrival time (minimum, maximum and mean), flow duration, and counters of the TCP flags above.

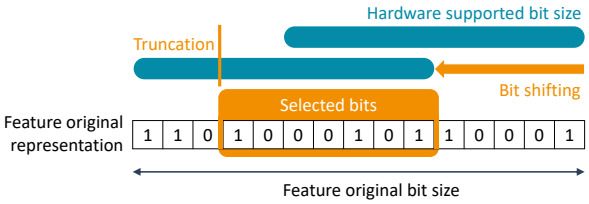


Figure 5: Hardware-aware tuning of feature representation.

An important remark is that, unlike previous works [17], *we do not use IP and MAC source or destination addresses as features*: the rationale is that such end host identifiers allow to artificially inflate the model performance in most use cases considered in the literature and in our work. Indeed, these use cases rely on measurement data collected in limited scenarios, and a sufficiently complex model fed with IP or MAC addresses can learn during training which end hosts are associated to the target classes (*e.g.*, belong to a specific type of device, or inject malicious traffic). Then, the inference task becomes unnaturally trivial in tests reproposing the same end hosts, as in most use cases adopted by the works presented in Section II-B. Yet, the model would spectacularly fail once deployed in the wild, where it would not find the same hosts it learned to recognize. By excluding identifiers, we force the model to complete the task based on inherent properties of the traffic, so that it generalizes to previously unseen hosts.

Feature engineering. We follow classical procedures for feature selection, by training an RF model with all available features, and obtaining a feature ranking according to the Mean Decrease in Impurity (MDI). We then train another set of models by adding the features one by one, to find the smallest subset of features that achieve performance comparable to or better than those of the full model with all the features.

What makes our feature engineering unique is that *we adapt the representation of the features to the specifications of the hardware*. More precisely, state-of-the-art RF mapping strategies use range matches to compare feature values against thresholds from the model, as explained in Section III-D. The size of range matches is bounded to a maximum number of bits in real-world equipment[8]. While this limit does not curb packet-level features rendered over a few bits, it affects flow-level ones, and in particular time-related features that are by default represented with floating point precision.

To cope with this issue, `Flowrest` fine-tunes the description of features exceeding the size limit, by (i) shifting their binary representation, and then (ii) truncating it to only consider an ideal bit length smaller than the limit. This procedure, illustrated in Figure 5, happens at training time via exhaustive search of the best combination of shift and truncation for each

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

feature. The result can be replicated in hardware during the flow-level feature handling phase, as it only uses basic binary operations. The fine-tuning ensures that all comparisons used by the RF model are compatible by design with the capabilities of the target programmable switch, and that training is optimized for performance in real-world equipment.

Model design. Jointly with the feature selection above, we perform a traditional grid search on the maximum number of trees t, and the maximum depth of the each tree d. Therefore, the feature ranking and processing described before is in fact derived for each RF of hyperparameters t and d. We compare the models with all (t, d) on the basis of the F1-score defined in Section IV-C, and pick the best performing one. The size of the model ultimately depends on the complexity of the use case: for instance, (t, d) varies from $(2, 3)$ to $(3, 11)$ in the use cases we consider in our experiments.

Importantly, *we also tailor the choice of hyperparameters in the RF model to the underlying hardware target*. Specifically, state-of-the-art RF mapping strategies such as that presented in Section III-D use ternary matches on codewords and associated masks to perform the final classification. This implies that the codeword length cannot exceed the maximum ternary matching size allowed by the actual Ternary Content Addressable Memory (TCAM) present in the switch.

We ensure that the constraint above is met by the model during the design phase. To this end, we leverage the fact that each tree in the RF generates one codeword, where every bit encodes one node of the RF, leaves excluded. As in a full binary tree with $n - 1$ total nodes we always have n leaves, we can control the codeword length by limiting the number of allowed leaves to the maximum ternary matching size in the programmable switch. We implement this design control by selecting the best nodes in each tree based on the relative reduction in impurity [42], until the target number of leaves is attained. Note that limiting the number of leaves does not bound the tree depth, which can span all the way to $d = n$, so that the model retains its flexibility.

IV. Experimental setup and results

We implement `Flowrest` in an experimental testbed, and run tests with multiple user-plane inference use cases. We expressly include tasks of unprecedented high dimensionality, so as to assess how the performance of a practical flow-level approach scale with the complexity of the inference problem.

A. Hardware setup

We validate our solution in a testbed with three Edgecore Wedge100BF-QS programmable switches equipped with an Intel Tofino BFN-T10-032Q chipset and 32 100GbE QSFP28 ports; the testbed is completed by servers with Intel 8-core Xeon processors at 2GHz and 48GB of RAM and QSFP28 interfaces, resulting in a full 100-Gbps platform. The switches run Open Network Linux (ONL) and Intel's Software Development Environment (SDE) version 9.7.0. We developed a Python controller on top of the Barefoot Runtime Interface (BRI) to automatically perform the initial configuration of the switch, including the mapping of RF models that are trained with Scikit-Learn libraries as described in Section III-E. We then use 100Gbps connections to inject traffic into the switch from the server, by replaying pcap traces via Tcpreplay [43]. Figure 7e portrays the testbed in a rackmount configuration.

B. Use cases

We employ the experimental platform to test `Flowrest` in multiple use cases employed for user-plane machine learning, and targeting varied tasks of (i) device identification, (ii) service classification, and (iii) anomaly detection. All use cases are supported by public measurement datasets.

UNSW-IoT [44] is a device identification use case based on measurement data for 28 Internet of Things (IoT) devices, collected in a living lab emulating a smart environment. The objective is identifying the type of IoT device generating each traffic flow by looking at statistical features of the data packets. We train the models on 15 days of data and test them on one. Note that we use this use case in two flavors, by trying to identify all 26 device types, or a subset of 16.

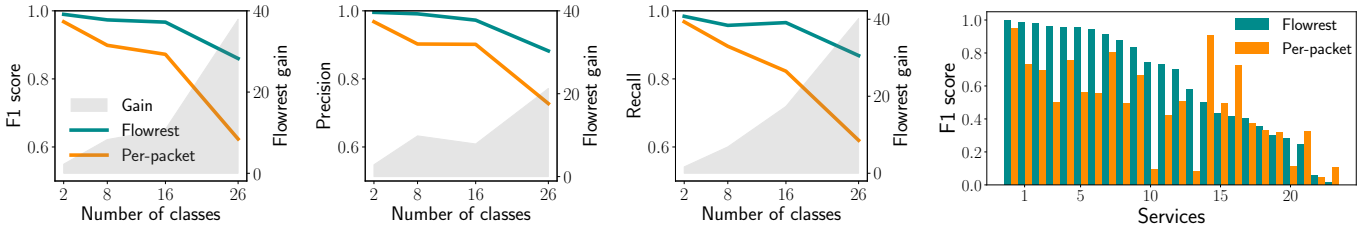
UNIBS-2009 [45], [46] is a traffic classification task based on real-world traces collected on the edge router of the University of Brescia campus network, capturing traffic from 20 workstations. The traces include web traffic (HTTP/HTTPS), mail (POP3, IMAP4, SMTP), peer-to-peer applications (BitTorrent, Edonkey) and other protocols (FTP, SSH). The goal is associating each traffic flow to one of 8 application categories. We use one day of traffic for training and a second for testing.

CICIDS2017 [47] is an anomaly detection use case based on measurements collected on a testbed at the University of Brunswick. The testbed includes two networks: a victim-network, which is a secure infrastructure with a set of computers running a daemon which implements benign behaviors; and an attack-network performing 7 types of attack. The goal is a binary classification of flows into benign and malicious. We use the Friday dataset, with a 75-25 split of train and test.

C. Benchmark and metrics

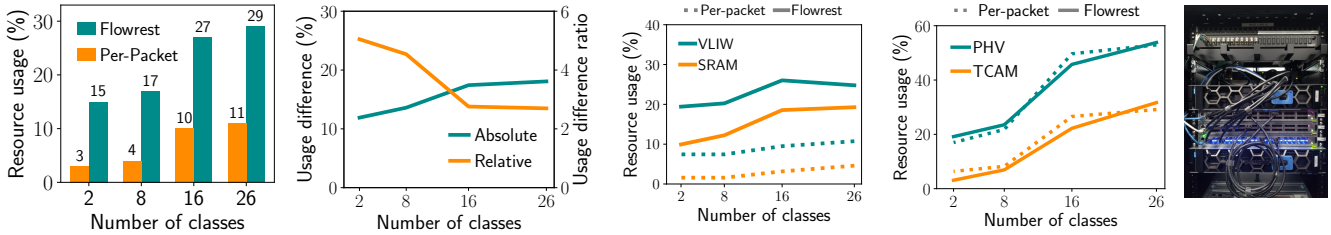
As explained in Section II-B, no hardware-compliant implementation for flow-level RFs exists today that we can use as a term of comparison. Therefore, we compare `Flowrest` against a state-of-the-art packet-level RF model. Specifically, we train dedicated RFs using only packet-level features with the Scikit-Learn libraries, and we implement them in hardware by pruning our framework from all its flow-level operations. This benchmark is a pure packet-level RF that also uses the efficient mapping described in Section III-D; it is thus representative of the performance of recent proposals like Planter [17] or Mousika [30]. We stress that this is the very first direct comparison of packet-level and flow-level RF models for programmable switches, and the fact that we run it with real-world equipment is a clear added value for the test.

The quality of `Flowrest` and the packet-level benchmark is assessed via classical metrics, applied across all use cases: these are (i) the precision, (ii) the recall, and (iii) their harmonic mean, *i.e.*, the F1 score.



(a) (b) (c) (d)

Figure 6: Performance of `Flowrest` and packet-level benchmark in terms of (a) F1 score, (b) precision, and (c) recall, across the four use cases. (d) Breakdown of the F1 score across the 26 classes of the UNSW-IoT use case.



(a) (b) (c) (d) (e)

Figure 7: Resource usage of `Flowrest` and packet-level benchmark in terms of (a) total average resources, (b) difference between the two models, (c) PHV and TCAM, (d) VLIW and SRAM. (e) Picture of the experimental testbed used in our study.

D. Results

Figure 6a shows the F1 score of `Flowrest` and the benchmark in the four use cases, ordered by their number of classes. `Flowrest` achieves good performance in all cases, with scores that start at 0.99 for the binary case and stay at 0.86 with 26 classes. The packet-level model does not scale equally well: it yields good performance in the simpler tasks, but the F1 score drops to 0.62 in the most complex inference problem. The accuracy gain of `Flowrest` over the per-packet benchmark is 2% for a binary classification, but grows with the number of categories up to 39% with the most challenging task we study.

Figures 6b and 6c tell apart the contributions of precision and recall, showing that a flow-level RF model helps improving in particular the recall. Figure 6d offers a breakdown of the F1 score per class, in the UNSW-IoT use case with 26 device types. Here, `Flowrest` yields a score above 0.85 in 9 classes, with another 6 classes above 0.50. While the other classes have F1 scores below 0.50, these are rare devices; instead, the performances are especially good for well-represented classes, as proven by the high aggregate F1 score in Figure 6a. When juxtaposing our model to the packet-level approach, and apart for a couple of outliers, `Flowrest` consistently outperforms the benchmark, in some cases by a very large margin.

However, our flow-level framework is much more structured than the minimum support needed for packet-level inference, and this induces a higher use of resources in the programmable switch. Figure 7a shows that `Flowrest` consumes 15% to 29% of the total average resources used by the baseline P4 program for core L2/L3 switching functions, *i.e.*, `switch.p4`. While this is a reasonable figure relative to that of a legacy forwarding configuration, the resource usage is still sensibly higher than in a per-packet approach where the same values range from 3% to 11%. Yet, when looking at how these numbers scale with the complexity of the task, interesting trends emerge: in Figure 7b, the absolute difference of usage is constant, and the relative resource consumption decreases.

The results point at the presence of a rather stable offset between flow-level and packet-level solutions, determined by the need of the former to implement structures capable to handle flows and their stateful features. The good news is that such structures appear to have a nearly constant cost that is not affected by the dimensionality of the inference task. This is corroborated in Figure 7c by the nearly fixed offset between the usage of, *e.g.*, SRAM that serves as the stateful memory for the flow management table, or Very Long Instruction Word (VLIW) that stores arithmetic instructions applied to flow-level features. Moreover, it is worth noting that `Flowrest` does not entail any added cost in terms of usage of expensive and scarce resources such as PHV and TCAM, in Figure 7d.

V. Conclusions

We proposed `Flowrest`, which we experimentally prove to set a new standard in flow-level in-switch inference. Our model yields a number of technical contributions and novel concepts that have general application to user-plane machine learning, like the synergy of user and control planes for effective line-rate inference, or the design of hardware-aware machine learning models. The authors have provided public access to their code and/or data at https://github.com/nds-group/Flowrest.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement no. 101017109 "DAEMON", which supported M. Gucciardo, and the Marie Skłodowska-Curie grant agreement no. 860239 "BANYAN", which supported A.T.-J. Akem. We thank the support of the Intel Connectivity Research Program.

REFERENCES

[1] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *CoRR*, vol. 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, Feb. 2022.

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

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

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

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

[7] The 5G Infrastructure Association (5G IA), "European Vision for the 6G Network Ecosystem," Jun. 2021.

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

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

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

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014.

[12] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNets'17*. NY, USA: ACM, 2017.

[13] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, p. 2–10, sep 2017.

[14] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *18th NSDI*. USENIX, Apr. 2021, pp. 785–808.

[15] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" *NetCompute '18*, 2018.

[16] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *HotNets 2019*. NY, USA: ACM, 2019, p. 25–33.

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

[18] K. Razavi, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "Distributed DNN serving in the network data plane," in *EuroP4 '22*. NY, USA: ACM, 2022, p. 67–70.

[19] G. Siracusano and R. Bifulco, "In-network neural networks," *CoRR*, vol. abs/1801.05731, 2018.

[20] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015.

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

[22] N. Corporation, "ConnectX SmartNICs - 10/25/40/50/100/200 and 400G Ethernet Network Adapters," 2022. [Online]. Available: https://www.nvidia.com/en-gb/networking/ethernet-adapters/

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

[24] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Practical in-network classification," *arXiv*, 2022.

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

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

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

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

[29] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[30] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation," in *IEEE INFOCOM 2022*, pp. 1938–1947.

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

[32] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic classification on the fly," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23–26, 2006.

[33] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.

[34] L. Bernaille and R. Teixeira, "Early recognition of encrypted applications," in *PAM*. Springer, 2007, pp. 165–175.

[35] M. Jaber, R. G. Cascella, and C. Barakat, "Can we trust the inter-packet time for traffic classification?" in *IEEE ICC*, 2011, pp. 1–5.

[36] G. Lu, H. Zhang, M. Qassrawi, and X. Yu, "Comparison and analysis of flow features at the packet level for traffic classification," in *ICCVE*, 2012, pp. 262–267.

[37] R. Panigrahy and S. Sharma, "Reducing TCAM power consumption and increasing throughput," in *Proceedings 10th Symposium on High Performance Interconnects*, 2002, pp. 107–112.

[38] H. Kim, X. Chen, J. Brassil, and J. Rexford, "Experience-driven research on programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, p. 10–17, 2021.

[39] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, 2019.

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

[41] "Wireshark." [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html

[42] F. Mola and R. Siciliano, "A fast splitting procedure for classification trees," *Statistics and Computing*, vol. 7, pp. 209–216, 1997.

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

[44] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, 2019.

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

[46] A. Este, F. Gringoli, and L. Salgarelli, "On-line SVM traffic classification," in *2011 7th IWCMC*, 2011.

[47] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," *ICISSP 2018*, pp. 108–116, 2018.