This is the author's accepted version of the journal article. The final version published is B. Bütün, D. de Andres Hernandez, A. Carré, M. Gucciardo and M. Fiore, "Distributing Inference in the User Plane of Complex Network Topologies with DUNE," IEEE/ACM Transactions on Networking, doi: TBD.

Distributing Inference in the User Plane of Complex Network Topologies with DUNE

Beyza Bütün, *Student Member, IEEE,* David de Andres Hernandez, *Member, IEEE,* Alexis Carré, Michele Gucciardo, *Member, IEEE,* and Marco Fiore, *Senior Member, IEEE*

***Abstract*—The deployment of Machine Learning (ML) models in the user plane has emerged as a promising approach to enable line-rate in-network inference, thereby reducing end-to-end latency and improving the scalability of network functions such as network telemetry. Nevertheless, integrating ML models into programmable switches remains challenging due to stringent memory and computational constraints. Prior work has predominantly focused on deploying monolithic ML models into individual programmable network devices, an approach fundamentally limited by hardware resources, particularly for complex classification tasks. In this paper, we introduce DUNE, a novel framework that, for the first time, enables distributed user-plane inference across multiple programmable network devices. DUNE employs original, fully automated techniques to (i) decompose large ML models into lightweight sub-models that retain inference accuracy while reducing resource consumption, and (ii) determine the design, sequencing, and placement of these sub-models to support efficient, distributed joint packet- and flow-level inference. We deploy P4 implementations of DUNE both in an experimental testbed with industry-grade programmable switches and in an emulation environment. We leverage the former setup to demonstrate the practical viability of the solution with real-world hardware in a simple linear topology, and the latter to evaluate DUNE 's scalability to larger and more complex network topologies coexisting with routing strategies. In both cases, we evaluate the framework using real-world traffic on two challenging classification tasks: our results show that DUNE not only reduces per-switch resource usage compared to traditional monolithic ML deployments but also improves inference accuracy by up to 7.5%.**

***Index Terms*—Programmable switches, machine learning, distributed inference, random forest, P4, mininet, bmv2.**

I. Introduction

Programmable data planes have rapidly evolved into a cornerstone of modern networking, enabling innovations that extend far beyond traditional packet forwarding. By allowing fine-grained control over packet-processing logic at line rate, user-plane programmability has fueled advances across a wide spectrum of network functions, including monitoring and telemetry, caching, intrusion detection, load balancing, and verification, as documented in recent surveys [1]–[3]. Beyond these traditional use cases, the advent of programmable network hardware has also opened the door to deploying Machine Learning (ML) models directly in the data plane for in-network inference. In this paradigm, models trained offline can be deployed into programmable switches or smart Network Interface Cards (smartNICs) and executed on every packet at line rate [4]. This capability enables inference to be carried out at the full forwarding capacity of modern devices (*e.g.*, on Tbps-scale traffic in switch ASICs) with extremely low latency (*e.g.*, in the order of tens of nanoseconds). An additional advantage is that performing inference within the data plane provides operators with instantaneous insights into packet-level traffic dynamics. Such real-time intelligence can substantially ease network planning by enabling more accurate capacity forecasting, adaptive traffic engineering, and proactive resource allocation, ultimately reducing the reliance on coarse offline measurements and reactive reconfiguration.

The benefits of performing inference in the user plane, compared to relying on control-plane ML, are significant. By avoiding cross-plane communication, user-plane inference achieves markedly lower delays (reduced by orders of magnitude), improved scalability (since overheads such as traffic mirroring are avoided), and lower costs (removing the need for specialized hardware, *e.g.*, for Deep Packet Inspection). At the same time, integrating ML models into programmable forwarding devices comes with notable challenges. These stem from the stringent constraints of the hardware—limited computational capabilities, scarce memory, and pipeline designs optimized for packet forwarding rather than ML workloads. Ultimately, these restrictions let user-plane ML pay a price in terms of inference accuracy as the task complexity grows [5].

Recently, a variety of solutions have been proposed to adapt ML models to user plane environments, carefully tailoring the design and mapping of the original ML architectures to the highly constrained programmable network hardware targets. However, almost all prior works, which are thoroughly reviewed in Section II, have pursued a *monolithic* deployment strategy, deploying a complete model within a single programmable device. This design, while practical for proof-of-concept demonstrations, inherently limits scalability and overlooks the distributed nature of real-world user planes, which span tens or even hundreds of switches and middleboxes. By constraining inference to a single device, these approaches restrict the achievable accuracy, flexibility, and robustness of in-network ML. Existing studies following this paradigm have explored inference in switches [6], possibly augmented with external

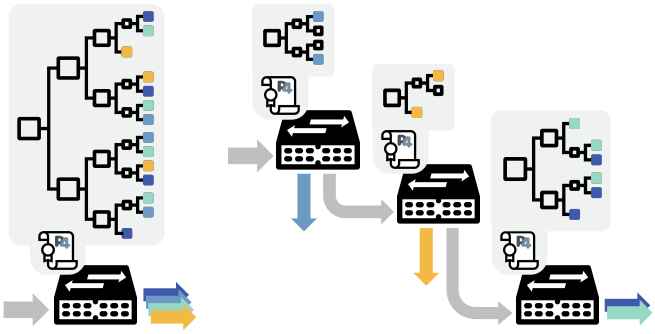
This research was supported by the SNS JU and the European Union's Horizon Europe research and innovation program under Grant Agreement No. 101139270 (ORIGAMI). B. Bütün is a predoctoral fellow (PIPF-2022/COM-24867), co-financed by Comunidad de Madrid.

B. Bütün and D. de Andres Hernandez are equal contributors to the work and co-first authors of the manuscript; both are with IMDEA Networks Institute, 28918 Madrid, Spain, and Universidad Carlos III de Madrid, 28911 Madrid, Spain. (email: {beyza.butun, david.deandres}@networks.imdea.org)

A. Carré is with École Normale Supérieure de Lyon, 69342 Lyon, France. (email: alexis.carre@ens-lyon.fr)

M. Gucciardo is with NEC Laboratories Europe, 28108 Madrid, Spain. (email: michele.gucciardo@neclab.eu)

M. Fiore is with IMDEA Networks Institute, 28918 Madrid, Spain. (email: marco.fiore@networks.imdea.org)



(a) Monolithic (b) Distributed

Figure 1: Conceptual illustration of (a) traditional monolithic and (b) proposed distributed user-plane inference strategies.

accelerators [7], and in smartNICs [8], using models such as Decision Trees (DTs) [9], Random Forests (RFs) [10], and Neural Networks (NNs) [11], operating on packet-level [12], flow-level [13], or at both levels jointly [5].

A much more flexible alternative is to *distribute* the ML model across multiple network devices, enabling inference to be carried out progressively as packets and flows traverse the network rather than being confined to a single point. This way, it balances resource usage across the network rather than overloading a single device, and increases resilience by preventing inference from becoming a single point of failure. Figure 1 shows this concept. In previous studies, the whole ML model (*e.g.*, a DT in the example) is coded as P4 programs and fitted into a single network device (*e.g.*, a switch in the example), abiding by its strict resource constraints. Our proposal is to *decompose* the operation of the original monolithic ML model into sub-models that (i) can be deployed as P4 programs into different devices and (ii) jointly execute the target inference task. As per the figure, this lets each device solve a portion of the problem (*e.g.*, identifying a specific traffic class) and leave the rest to the downstream network. Such a distributed operation is inherently aligned with the notion that packets do not traverse a single appliance in the network; rather, they are naturally forwarded across multiple devices that can thus jointly perform the target inference task.

Although the idea is conceptually straightforward, its practical realization raises several non-trivial challenges. Key challenging and open questions include how to decompose the original ML model in a way that preserves inference accuracy while keeping the resource footprint of each sub-model manageable, how to determine an optimal ordering of the resulting sub-models, and how to select their optimal placement on the predefined network paths in a given topology that ensures the target traffic traverses all sub-models in the correct order. Moreover, the decomposition process itself should ideally be automated and generalizable to arbitrary inference tasks. Furthermore, the deployment of distributed inference also introduces system-level challenges, such as ensuring scalability across diverse network architectures and achieving seamless integration with existing control and forwarding mechanisms.

In this paper, we present DUNE[1], the first practical framework for the execution of distributed user plane inference in real-world programmable hardware. DUNE focuses on a specific type of inference, *i.e.*, classification, since this is the relevant task in the vast majority of network functions that include traffic classification, intrusion detection, application identification, spam filtering, bandwidth management, and congestion management. The design, implementation, deployment, and evaluation of DUNE yield the following main contributions.

- We propose a novel framework to decompose large ML models trained to solve complex inference tasks into simpler sub-models that jointly perform the same operation. The framework is fully automated, can be applied to any class of input ML model, and solves original optimization problems in order to preserve the inference accuracy while minimizing the complexity of the sub-models.
- We present a design for the ML sub-models and their interactions that enables distributed packet- and flow-level inference across multiple programmable network devices. To orchestrate this process, we introduce a state machine that specifies the operations executed within each switch and the state transitions they trigger.
- The sub-models are deployed across a given network topology with varying levels of complexity, utilizing a pre-established routing strategy, while preserving compatibility with standard network functionalities. To this end, we adapt the Planner of DINC [15] to our specific design so as to determine optimal deployment strategies for sub-models along predefined network paths.
- We implement DUNE using the P4 language in both emulated (*i.e.*, via the bmv2 framework [16]) and hardware (*i.e.*, in Intel Tofino switches) targets. We release the full software and hardware implementations along with all experimental artifacts as open source to ensure transparency, verifiability, and reproducibility of our research [17].
- We run proof-of-concept experiments in a real-world industry-grade testbed with a linear topology, as well as large-scale tests in Mininet [18] across a range of complex network topologies. Results with two challenging classification tasks based on measurement traffic traces demonstrate the feasibility of our approach in practical scenarios and its gains in terms of higher accuracy over traditional single-device monolithic approaches.

II. Related work

User-plane inference, also referred to as in-network ML, was postulated as a viable paradigm only a few years ago [12], [19], [20], and has since attracted growing attention from the research community. The field is now very active, and even a recent first survey published this year [4] does not capture the substantial innovation that occurred in the past six months.

Solutions to deploy trained ML models in programmable network hardware have considered different targets, including

[1]A preliminary version was presented at IEEE INFOCOM 2025 [14].

network accelerators [7] and SmartNICs [8], yet programmable switches stay the preferred space for user-plane inference and the focus of all research summarized next.

Monolithic tree-based models. The vast majority of the literature proposes models based on tree structures, *i.e.*, DTs or RFs, that are implemented in individual switches. Early solutions deal with the efficient mapping of trees to the switch Application Specific Integrated Circuit (ASIC) [6], [21], [22], considering individual DTs [23]–[26] or special-purpose solutions tailored to one inference task [27], [28]. Subsequent works cope with the limited switch resources through knowledge distillation approaches [9], multi-level flow table representations [29], or flattening and multiplexing of sub-trees across ALUs [30]. Other enhancements include exploiting both ingress and egress stages of the switch ASIC [31], providing support for inference based on flow-level features [13], or considering more complex tree-based models [10]. Moreover, recent efforts achieve joint packet- and flow-level inference by automatically selecting the best option based on stateful information about each traffic flow [5], [32]. All these works invariably aim at deploying monolithic ML models that perform all inference operations, including feature extraction, state maintenance, tree traversal, and consensus resolution, in a single programmable switch.

Monolithic neural networks. Neural architectures are significantly more involved than DTs or RFs, and embedding them in programmable switches has proven especially challenging. Still, after early attempts [33], several breakthroughs have led to workable implementations that rely upon custom activation functions [34] or Recurrent Neural Network (RNN) architectures adapted to the limited data plane stages [11]. Recently, also compressed and restructured Convolutional Neural Network (CNN) models have been successfully demonstrated in programmable network hardware [35]. Again, all these approaches propose to implement one NN in one switch.

Distributed ML. There exist several ways in which user-plane ML-driven inference can be distributed. In a first model, *inference is distributed across planes*, with part of the process run in programmable network equipment and part in a more powerful control-plane server; several works have explored this direction [36]–[38] but all assume that user-plane operations happen in the same device. A second approach is the *separation of feature extraction from the rest of the inference process*, with the former run in different devices than the latter; the relevant position paper [39] still considers the ML models to be monolithic and deployed in single switches.

The third acceptation is that of our interest, and intends to *distribute the operation of the ML model itself across multiple user-plane devices*, as depicted in Figure 1b. To the best of our knowledge, there are just two solutions proposed to date in this direction, *i.e.*, NetNN [40] and MUTA [41]. NetNN deploys a NN across a programmable network by implementing different layers of the neural architecture in subsequent switches. However, it realizes a specific neural model with a precise sequence of convolutional and dense layers trained for one task, *i.e.*, intrusion detection; instead, we seek a general-purpose solution that can distribute any ML model across a given programmable network. Moreover, NetNN is only emulated with the bmv2 behavioral model that is known to simplify many of the limitations of the real-world ASICs, and may result in performance losses and a need for partial model re-design when porting the solution to actual hardware [42]. MUTA targets instead network management by jointly addressing multiple tasks through a shared multi-task learning (MTL) model instead of deploying separate models for each task. MUTA builds an MTL network that shares layers across tasks and maps them onto programmable switches for in-network inference. However, its scalability in the number of layers is bounded by the traversed switches, and the complexity of each layer is severely curbed by the switch resources.

Dataplane Disaggregation. The concept of dataplane program disaggregation was introduced by Flightplan [43], a toolchain that splits a P4 program into subprograms that can be executed across devices. This approach allows for the combination of heterogeneous hardware to overcome the limitations of single-target execution and efficient resource usage. While Flightplan provides a framework for automated splitting, placement, and runtime support, the splitting task relies on a manual segmentation approach. Automating the splitting task is not straightforward nor easily generalizable, as each NF type might require specific segmentation procedures, constraints, and goals. More recently, DINC [15] has also addressed the problem of distributed in-network computing. Like Flightplan, DINC also relies on manual markers to identify the program segments, but, unlike Flightplan, which focuses on single-path scenarios, DINC considers multiple network paths and aims to ensure service provisioning for any set of predefined routes.

Progress beyond the state of the art. We present DUNE, a solution that can automatically disaggregate any ML model for classification so that it can be deployed over a programmable network. Our solution is practical, as its design is tailored to real-world hardware and its operation is demonstrated with industry-grade programmable switches. We stress that DUNE goes beyond Flightplan [43] or DINC [15], as it also produces the task *segmentation* that is an input to such solutions.

III. Distributed ML Training with DUNE

As any other solution for user-plane ML [5]–[34], DUNE is composed of an off-line phase for the design and training of the ML model that is run in the control plane, and an on-line phase where the trained model is deployed in the programmable network and performs line-rate inference on the traffic. We present in this section the first phase, and in Section IV the second.

A. Workflow overview

The high-level workflow of DUNE during the off-line phase is outlined in Figure 2. Here, the goal of the framework is to generate a set of ML sub-models that (i) are compatible with the programmable network constraints and (ii) jointly execute the desired inference task in the target network.

To this end, historical measurement data about the target inference task, suitably labeled for supervised learning, is initially fed to a first ML model training stage ①. At this stage,

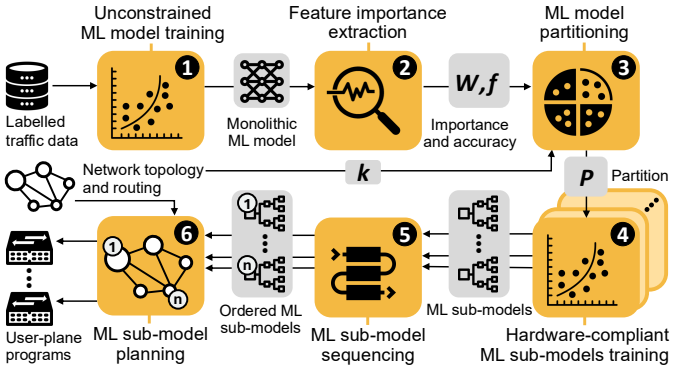


Figure 2: Overview of the DUNE workflow for the training of distributed ML compatible with programmable user planes.

the goal is to train the most accurate ML model possible, ignoring constraints from the programmable network hardware, and the framework can accommodate any ML paradigm, *e.g.*, deep neural architectures, support vector machines, or tree-based models, as expounded in Section III-B. The output of stage ① is a trained, complex, monolithic model that maximizes accuracy for the target traffic analysis task.

In stage ②, the monolithic ML model is examined so as to extract values that describe the importance of each input feature (*e.g.*, packet header fields or flow-level statistics) to estimate every model output (*e.g.*, a class of network traffic). Different tools can be used to that end, as outlined in Section III-C, producing an importance matrix $\boldsymbol{W}$ that expresses the relevance of features in explaining output variables and an accuracy vector $\boldsymbol{f}$ of the inference quality for each class.

The information in $\boldsymbol{W}$ and $\boldsymbol{f}$ is used in stage ③ to compute a partitioning $\boldsymbol{P}$ of the original inference task into sub-tasks. Each sub-task employs a subset of the input features to estimate a specific disjoint subset of the output variables, and the partitioning aims at creating groups of variables that can be well explained by a compact set of features. Crucially, this partitioning is bounded by a parameter k, derived from the given network topology and routing paths, which limits the maximum number of allowable sub-tasks to ensure compatibility with the available programmable devices. To identify the best partitioning, DUNE hinges upon an original optimization problem solved via an efficient heuristic, as per Section III-D.

For each sub-task in the partition $\boldsymbol{P}$, stage ④ then trains a dedicated ML sub-model that predicts the sub-task variables using the sub-task input features. In this case, the ML sub-models are designed abiding by the user-plane hardware constraints so as to ensure their compatibility with the target programmable network, as described in Section III-E.

As the ML sub-models tackle different portions of the whole inference task on the same network traffic, they must be run in sequence to execute the original function, similar to the chaining of Virtual Network Functions (VNFs). However, chaining ML sub-models propagates inference errors, which makes it critical to account for their (diverse) reciprocal accuracy when establishing an order of execution. As an intuitive example, it is critical to avoid deploying at the entry of the chain an ML model with a high rate of false positives: this leads to a large number of early misclassified flows, which are then never received by the downstream sub-models intended to infer the true category of such flows. As discussed in Section III-F, the stage ⑤ of DUNE solves a dedicated optimization problem based on the reciprocal accuracy of the sub-models to identify the order yielding the highest inference performance.

In the last stage, ⑥, the ordered sequence of ML sub-models is combined with information about the target network topology and traffic routes in order to decide how to deploy sub-models across the available switches. Specifically, the network topology and routing information are encoded as a set of available paths (*e.g.*, sequences of switches) that traffic flows may traverse. Then, an Integer Linear Program (ILP) inspired by the DINC Planner [15] is formulated to place sub-models to ensure that each and every packet is forwarded through sub-models in the correct order, while minimizing the number of switches where inference occurs–hence the overall user-plane resource consumption of the ML process. Full details are provided in Section III-G.

Finally, each ML sub-model is coded as a P4 program and injected into the target programmable user-plane devices. It is worth noting that the implementation of the sub-models into resource-constrained hardware targets is an entangled task, exacerbated by the fact that such sub-models must operate in a coherent way once distributed across any network topology and in the presence of packet losses or reordering. This involves several important design decisions and a dedicated operational logic, which are discussed in Section IV.

B. Unconstrained ML model training

Stage ① consists of the definition, hyper-parametrization, and training of a single ML model that solves the desired inference task. As anticipated, this model is *unconstrained*, *i.e.*, is not limited by the programmable hardware memory or compute constraints; hence, it can be as complex as the control-plane ML Operations (MLOps) resources allow. The exclusive objective of this unconstrained ML model is achieving the maximum accuracy possible in the target inference task.

While this stage of DUNE is general and can accommodate different ML paradigms, in our implementation we opt for using a large RF as the unconstrained model. The motivation is twofold: first, RFs are inherently more explainable than neural architectures, making the extraction of feature importance from a trained model much faster and more precise, as explained in Section III-C; second, for the challenging traffic classification use cases employed to evaluate the performance of DUNE and presented later in Section VI, using NNs based on Multi-Layer Perceptron (MLP) as the unconstrained ML models returned sensibly lower accuracy than that of RFs.

To design the unconstrained RF, we start from the RF model recently proposed by Jewel [5], a tree-based architecture that can perform packet-level inference (*i.e.*, using only features extracted from the header of the current packet, such as payload length) on the first packets of a flow and automatically switch to a more accurate flow-level inference (*i.e.*, using features that relate to all received packets in the same flow, such as the average inter-arrival time) as soon as enough

packets in the flow have been seen to build reliable flow statistics. Note that we consider the monolithic version of this type of tree that is originally implemented in Jewel as a benchmark in the comparative evaluation in Section VII.

DUNE hyper-parametrizes the joint packet- and flow-level RF model by training it with an exhaustive grid search over the space of hyper-parameters, *i.e.*, the maximum number of trees, the maximum depth of each tree, and the rank of the packet for which the flow-level inference is triggered. The grid search is iterated so as to also identify the optimal set of input features that maximizes the unconstrained model accuracy: specifically, we start from a model trained with the full set of all available features, compute the Mean Decrease in Impurity (MDI) to remove the least relevant feature, re-run the grid search, and iterate. In the end, we select the RF model that achieves the highest *macro F1* score (see Section VI-D for a definition).

C. Feature importance extraction

Stage ② extracts the relationships between input features and output variables from the unconstrained model produced by the previous stage. It is worth highlighting that here we do not seek to rank the importance of the input features for the inference task as a whole, for which standard techniques such as the MDI used in stage ① exist. Instead, we need to quantify the explanatory power of each input feature for each individual output class, which is a sensibly harder problem.

DUNE takes advantage of the design choice of adopting an RF model in stage ① to extract feature importance efficiently. Specifically, it leverages the Per-Class Feature Importance (PCFI) method [44], which operates on tree structures and extends MDI by considering the paths (*i.e.*, sequences of nodes from the root node to a leaf that contains the inference decision) that compose the trained RF. For each path in every RF tree, PCFI calculates the importance of a given feature as the total MDI at nodes that split their sub-trees based on that feature; it then associates such an MDI value to the class in the leaf at the end of the path. Once all paths have been processed, PCFI computes the final importance values for a feature-class pair as the mean of all MDI values of the input feature associated with the target class.

The PCFI approach is not the only one possible. For instance, the well-known SHapley Additive exPlanations (SHAP) [45] offer an alternative way to derive from a trained model how much each input feature contributes to the inference of an individual output variable. SHAP computes so-called Shapley values for each feature and produced inference output by evaluating all possible permutations of the features and determining the marginal contribution of the feature to the prediction as the difference caused by its absence. While SHAP has the advantage of being applicable to any ML model, it suffers from poor exponential scalability in the model complexity [46] that makes it impractical in many cases.

In the case of tree-based models, a compute-efficient variant of SHAP exists: TreeSHAP [47] leverages the structure of trees to compute exact Shapley values in polynomial time. Yet, the complexity of PCFI and TreeSHAP is still very different. Let us denote by $\mathtt{t}$, $\mathtt{L}$, and $\mathtt{d}$ the number of trees, the maximum number of leaves in a tree, and the maximum depth of each tree in the unconstrained RF model, respectively: then PCFI has a complexity $O(\mathtt{t}\cdot\mathtt{L}\cdot\mathtt{d})$ whereas that of TreeSHAP is $O(\mathtt{t}\cdot\mathtt{L}\cdot\mathtt{d}^2)$. As tests with the reference use cases in Section VI show similar accuracy in the feature importance returned by PCFI and TreeSHAP, we opt for the former in our implementation.

D. ML model partitioning

Stage ③ breaks down the original inference task into a series of smaller sub-tasks that jointly achieve the same goal. To this end, DUNE solves an original Set Partitioning Problem (SPP) to identify sub-tasks as disjoint groups of output classes that can be explained with high accuracy by a compact set of input features each. We next formalize SPP as an optimization problem and present the efficient heuristic used to solve it.

1) Formulation: Let $V = \{v_1, \ldots, v_m\}$ be the set of all m output variables (*i.e.*, classes) that must be explained with the set of r features in the inference task. Also, let us denote by $S = \{S_1, \ldots, S_n\}$ the set of all possible blocks (*i.e.*, class groups). $P \subset \{1, \ldots, n\}$ is a partition of V if and only if

$$\bigcup_{i \in P} S_i = V \quad \text{and} \quad S_i \cap S_j = \emptyset \quad \forall i, j \in P, i \neq j. \tag{1}$$

The goal of the SPP is to find the partition P that best balances inference accuracy and programmable network resource use. This translates into the following two requirements.

- (i) *Compactness*. P shall be formed by the smallest possible number of blocks. This requirement stems from the fact that each block will be later mapped by our framework into a separate ML sub-model, and every added sub-model yields overhead (as it requires, *e.g.*, maintaining its own flow state registers or mapping its ML logic to a user-plane device) and grows the number of network devices required to implement the full distributed solution.
- (ii) *Effectiveness*. Each block in P shall maximize the accuracy of inference for the variables it includes by using a minimum number of features. The rationale is that accuracy must be preserved with limited feature sets, which later translates into less complex ML sub-models.

It is worth noting that the two requirements above entail a clear trade-off: the best accuracy using small feature sets is achieved with tiny blocks each dedicated to very few variables; yet, that also creates a large number of ML models, which ultimately harms resource usage. The SPP thus aims at finding the best partition that solves such a trade-off.

The optimization problem can be formalized using a matrix representation. Let $\boldsymbol{s}_i \in \mathbb{Z}_2^m, i \in \{1, \ldots, n\}$ be an indicator vector designating which variables in V are part of a block S_i, *i.e.*, the k-th element of $\boldsymbol{s}_i$ is 1 if class k is in S_i and 0 otherwise. We then construct a matrix $\boldsymbol{A} \in \mathbb{Z}_2^{m \times n}$ with the $\boldsymbol{s}_i$ vectors as the n columns. By defining a vector of binary decision variables $\boldsymbol{x} \in \mathbb{Z}_2^n$ whose i-th element indicates if a block S_i is selected as part of the output partition P, we can write the constraints in (1) as $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{1}$. The formal definition of the optimization problem is then

$$\max_{\boldsymbol{x}} \ \frac{1}{c(\boldsymbol{x})} \cdot \boldsymbol{g}^T \boldsymbol{x} \qquad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{1},\ \boldsymbol{1}^T \boldsymbol{x} \leq k, \tag{2}$$

where $c(\boldsymbol{x})$ captures requirement (i) above and denotes the cost of partitioning the inference task into the number of blocks defined by $\boldsymbol{x}$, whereas $\boldsymbol{g}^T\boldsymbol{x}$ models requirement (ii) as the accuracy achieved for the blocks in $\boldsymbol{x}$ with minimum features. Finally, $k \in \{2, \ldots, m\}$ is an optional upper bound on the number of blocks in the partition, reflecting a deployment constraint on the number of available programmable devices. When $k = m$, the constraint is trivially satisfied for any partition and the problem reduces to the unconstrained case.

More precisely, the cost $c(\boldsymbol{x})$ is expressed as

$$c(\boldsymbol{x}) = \frac{m-1}{m - \mathbf{1}_n^T\boldsymbol{x}}, \tag{3}$$

where $\mathbf{1}_n \in \mathbb{Z}_2^n$ is a vector of 1's; hence $\mathbf{1}^T\boldsymbol{x}$ is a scalar corresponding to the number of non-zero elements of $\boldsymbol{x}$, *i.e.*, the number of blocks in the selected partition. This entails a linearly decreasing advantage $1/c(\boldsymbol{x})$ as the number of blocks grows up to the total number of variables m.

As far as the second factor of the objective function in (2) is concerned, $\boldsymbol{g} \in \mathbb{R}^n$ is a vector of the gains g_i associated with all of the possible blocks S_i, which is expressed as

$$g_i = \Theta(\boldsymbol{s}_i) \cdot \Psi(\boldsymbol{s}_i), \quad \text{where} \tag{4}$$

$$\Theta(\boldsymbol{s}_i) = \max_{\boldsymbol{\phi}} \; \boldsymbol{s}_i^T\boldsymbol{W}\boldsymbol{\phi} - (\tfrac{1}{r}(\mathbf{1}_n^T\boldsymbol{s}_i)\mathbf{1}_r)^T\boldsymbol{\phi} \tag{5}$$

$$\Psi(\boldsymbol{s}_i) = 1/(1 + \max_{v_k|\boldsymbol{s}_i(k)=1} f_k - \min_{v_k|\boldsymbol{s}_i(k)=1} f_k). \tag{6}$$

In the expression in (5), the matrix $\boldsymbol{W} \in \mathbb{R}^{m\times r}$ contains the per-class feature importance values returned by stage ②, while $\boldsymbol{\phi} \in \mathbb{Z}_2^r$ is an array denoting a feature subset, *i.e.*, the h-th element is 1 if feature h is in $\boldsymbol{\phi}$, and 0 otherwise. Then, $\boldsymbol{s}_i^T\boldsymbol{W}\boldsymbol{\phi}$ is the cumulative importance retained by the selected features in $\boldsymbol{\phi}$ for classes in S_i, whereas $(\frac{1}{r}(\mathbf{1}_n^T\boldsymbol{s}_i)\mathbf{1}_r)^T\boldsymbol{\phi}$ is a penalty on the number of features used to achieve such importance.

In (6), f_k is the k-th element of array $\boldsymbol{f} \in \mathbb{R}^n$ provided by stage ② and denotes the accuracy (*e.g.*, the F1 score defined in Section VI-D) achieved by the unconstrained ML model in inferring class v_k. The expression introduces a penalty proportional to the maximum difference in accuracy for two variables in the block S_i: high differences indicate inherently diverse complexity of the inference process for the corresponding classes, which in turn suggests that such classes need ML sub-models of sensibly different size and are thus not good candidates for grouping.

2) Solution: The SPP is known to be NP-hard even in its basic optimization problem formulation where the objective function is linear in the decision variables $\boldsymbol{x}$. Yet, in that case, the SPP can be rendered as an Integer Linear Problem (ILP) and solved using standard solvers up to a reasonable size. Unfortunately, the objective function in (2) is non-linear, impeding such a simplification and making an optimal solution computationally impractical.

We propose a greedy approach, summarized in Algorithm 1, to approximate (2). The algorithm mimics an agglomerative clustering process where the gain in (4) is used as the *similarity metric* to merge at every iteration the two (blocks of) variables with maximum similarity. The algorithm performs $m - 1$ iterations (line 2); at each iteration, it computes gains[2] among all current blocks (lines 4–6), identifies the pair of blocks with maximum gain (line 7), and updates the blocks data structure with the new merged block (lines 8–15). When the partition is feasible (*i.e.*, $k = \emptyset$ or $|\textit{blocks}| \leq k$), its total gain is computed using (2) and possibly stored as the best solution (lines 16–22). When k is specified, the gain-maximization is restricted to partitions with at most k blocks, thus enforcing the cardinality constraint. Such gains are not monotonic in general across iterations, but take different shapes depending on the target inference task. Figure 3a shows the evolution of the gain matching our objective function (2) over iterations, for one of the reference use cases we will present in Section VI.

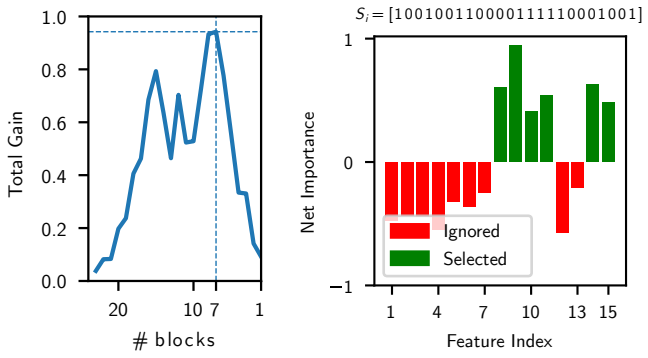
Algorithm 1 SPP greedy algorithm.

Require: $k \in \{2, \ldots, m\} \cup \{\emptyset\}$
1: $n^* \leftarrow m$, *gain-max* $\leftarrow 0$, (*blocks*) $\leftarrow$ all single-element blocks
2: **while** $|\textit{blocks}| > 1$ **do**
3: *tuples* $\leftarrow$ COMBINATIONS(*blocks*, 2)
4: **for all** *tuple* $\in$ *tuples* **do**
5: *gains* $\leftarrow$ COMPUTEBLOCKGAIN(*tuple*)
6: **end for**
7: *best tuple* $\leftarrow argmax(\textit{gains})$
8: **for all** *block* $\in$ *blocks* **do**
9: **for all** *element* $\in$ *best-tuple* **do**
10: **if** *element* $\in$ *block* **then**
11: REMOVE(*element*, *blocks*)
12: **end if**
13: **end for**
14: **end for**
15: ADD(*best-tuple*, *blocks*)
16: **if** $k = \emptyset$ **or** $|\textit{blocks}| \leq k$ **then**
17: *gain* $\leftarrow$ COMPUTEPARTITIONGAIN($blocks$)
18: **if** *gain* $>$ *gain-max* **then**
19: $n^* \leftarrow |\textit{blocks}|$
20: *gain-max* $\leftarrow$ *gain*
21: **end if**
22: **end if**
23: **end while**

The computation of the gain in (4) needs to be especially efficient since it is profusely used in Algorithm 1; hence, we propose a computationally viable implementation with cost $O(r)$. The gain is a product of (6) and (5): while the factor in (6) is inexpensive, we decompose the calculation of (5) into r sub-problems leveraging its structure. In each sub-problem, a feature is selected if the importance gain, *i.e.*, the minuend in (5), overcomes the uniform penalty, *i.e.*, the subtrahend in (5). For example, Figure 3b shows in green the features with a positive net importance gain for a given block s_i of the SPP for the same inference task of Figure 3a.

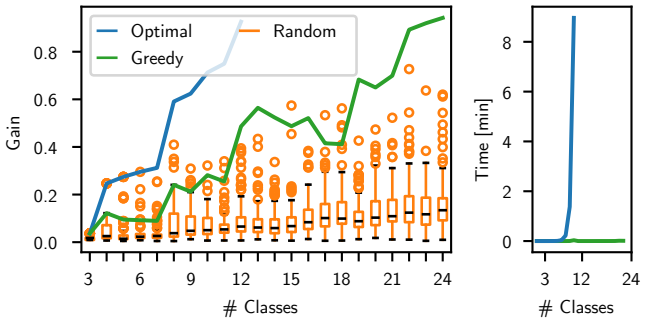
Figure 4 provides a visualization of the accuracy-complexity trade-off of the greedy algorithm solving the SPP. We consider the same inference task as in Figure 3, but limiting it to a given number of classes, shown on the abscissa of the plots: when considering, *e.g.*, 9 classes, we randomly select 9 of the 24 classes that constitute the problem and run the greedy algorithm to identify a good partition of such 9 classes. Figure 4a shows that our proposed heuristic (green) performs substantially better than stochastic partitions (orange) with a

[2]Gains calculated at a previous iteration are cached and not re-computed.



(a) Blocks selection (b) Feature selection

Figure 3: Greedy solution to the SPP in (2) in the UNSW use case. The plots show (a) the gain versus the number of blocks ($n* = 7$) and (b) the feature selection for a given block S_i.



(a) Gain performance (b) Execution time

Figure 4: SPP algorithms comparison over the UNSW dataset.

gain that tends to increase as the complexity of the SPP grows. The optimal solution (blue) clearly outperforms the greedy solution, yet Figure 4b shows that it also has an exponentially growing execution time that does not allow solving the SPP for more than a handful of classes. Instead, the greedy algorithm has a reasonable computational cost $O(m^2)$ in Figure 4b.

E. Hardware-compliant ML sub-model training

Stage ④ trains one ML sub-model for each block in the partition P^* output above. Unlike the unconstrained model trained in stage ①, these sub-models are designed to comply with the constraints of the target programmable network devices. Each sub-model aims at classifying the variables (*i.e.*, classes) present in the corresponding block, plus one additional *others* category that gathers all classes that are part of the total inference problem but are not included in the current block.

Our implementation of DUNE relies on RFs to realize each ML sub-model. This choice is grounded in the significant success that RFs had as a reference ML paradigm for user-plane inference, as reported in Section II. Nevertheless, DUNE's workflow is also compatible with other ML model types. Recent frameworks [48]–[50], that have successfully demonstrated the deployment of deep learning models on the data plane, confirm that they can also serve as sub-models within DUNE. The model design, hyper-parametrization and training are thus similar to those described in Section III-B and rely on a joint packet- and flow-level inference model. The difference is that the bit representation of the features, the maximum number of trees, and the maximum number of leaves per tree are now constrained to abide by the limitations of user-plane hardware. Indeed, the limited size of register entries, the small amount of memory, and the finite number of stages in programmable devices force restrictions that must be accounted for at the design stage to ensure the compatibility of the ML models with the network equipment. Since we target a user plane composed of programmable switches, we adopt the techniques introduced in Flowrest [13] to (i) engineer features and (ii) limit tree dimension, thus ensuring that the design of the ML sub-models is fully compatible with the network hardware.

The best RF sub-model for each block is identified from the grid search over the (constrained) hyper-parameters by balancing accuracy and resource usage. Specifically, we select the RF that maximizes $\alpha \cdot \text{F1} + (1-\alpha) \cdot \text{U}_{\text{TCAM}}$, where F1 is the *macro F1* score defined in Section VI-D, U_{TCAM} is the expected usage of Ternary Content-Addressable Memory (TCAM) in the switch, and α is a tunable parameter that we set to 0.5 in our experiments to fairly balance the two contributions.

F. ML sub-model sequencing

Stage ⑤ orders the ML sub-models so as to optimize the inference performance. The sub-model sequencing impacts the distributed inference accuracy due to the following aspects.

- Confusion between sub-models addressing different parts of the global inference. The problem arises due to false positives from an upstream sub-model that generates incorrect early decisions on flows that in fact belong to the target classes of downstream sub-models.
- Diverse accuracy of each sub-model. Sub-tasks have non-uniform complexity, and each sub-model has a different error rate that affects all downstream sub-models.

The goal of this stage is then to generate an ordering such that ML sub-models with a low rate of false positives and a high accuracy are deployed earlier in the sequence. DUNE solves a simple optimization problem to that end. The objective is

$$\min_{\boldsymbol{X}} \sum_{i,j \in P^*,\, i \neq j} p_{ji} \cdot (1 - f'_i) \cdot x_{ij}, \quad x_{ij} \in \boldsymbol{X} \tag{7}$$

where p_{ij} is the rate of false positives[3] that the ML sub-model of block S_i generates with respect to classes in block S_j of the partition P^*, f'_i is the *macro F1* score of the sub-model for block S_i trained in Stage ③, and x_{ij} are binary decision variables that take value one if the ML-model of block S_i precedes that of block S_j in the ordering.

In order to ensure that each ML sub-model is executed only once in a valid order, we define the following two constraints.

$$x_{ij} + x_{ji} = 1, \quad \forall i, j \in P^*, i \neq j, \tag{8}$$

$$u_i - u_j + x_{ij} \cdot \|P^*\| \leq \|P^*\| - 1 \quad \forall i, j \in P^*, i \neq j, \tag{9}$$

[3]Since each ML sub-model is trained on the entire set of classes, it is possible to calculate the false positives across blocks. For instance, for the sub-model associated to block S_i we can compute the rate of flows belonging to the classes in block S_j that such a model misclassifies as its own.

where u_i is an auxiliary variable associated with the block S_i used to enforce the order of deployed sub-models and $\|P^*\|$ is the total number of blocks. The problem in (7)–(9) is a version of the well-known Traveling Salesman Problem and can be formulated as an ILP and solved efficiently.

G. ML sub-model planning

The ordering returned by stage ⑤ directly allows deploying the ML sub-models within a linear sequence of user-plane devices, so that the inference process can be distributed, *e.g.*, along a single, specific, predefined path. However, routes in a real-world network are not independent sequences of disjoint sets of switches; they are instead sets of intertwined paths resulting from traffic-engineering decisions on the underlying physical network topology. A trivial solution is then deploying the complete sub-model sequence along all possible paths in the network, which would, however, cause unnecessary duplication of functions, rapidly deplete switch resources, and ultimately fail to scale beyond tiny topologies.

Stage ⑥ determines the optimal deployment strategy for a given combination of network topology and routing, ensuring that all flows forwarded through the target network traverse all ML sub-models in the correct sequence. The goal is to ensure this condition while minimizing sub-model duplication across the network. To this end, we devise a target-agnostic deployment strategy inspired by the planner component of the DINC framework [15].

For a given network topology and routing, the deployment strategy we develop aims to achieve two objectives: (i) minimize sub-model duplication across paths, and (ii) maintain the execution order of sub-models. To this end, we formulate an ILP with an objective function

$$\min \sum_{s \in S} \sum_{m \in M} X_{s \leftarrow m}, \tag{10}$$

where $M := \{1, \ldots, N_M\}$ is a linearly ordered set of sub-models obtained as explained in Section III, $S := \{1, \ldots, N_S\}$ is a set of switches in a given topology, and $x_{s \leftarrow m}$ is a binary variable indicating whether sub-model m is deployed on switch s, *i.e.*, $x_{s \leftarrow m}$ is 1 if the sub-model m is deployed on the switch s, 0 otherwise. The objective minimizes the total number of sub-models deployed across the network. We then ensure that each sub-model is present at least once on every path by the following constraint

$$\sum_{s \in P} X_{s \leftarrow m} \geq 1 \quad \forall m \in M, \forall P \in \mathcal{P}, \tag{11}$$

where $\mathcal{P} := \{P_i\}_{i=1}^{N_p}$ is a set of paths in the given topology, and each path $P_i \in \mathcal{P}$ is an ordered sequence of switches. Moreover, we need to guarantee the correct execution order of sub-models. For any two sub-models $m < m'$, if m' is deployed on a switch at position j along a path P_i, then m must be deployed on a switch that appears earlier on the same path. Formally, this results in a constraint

$$\sum_{k=1}^{j-1} X_{s_i^k \leftarrow m} \geq X_{s_i^j \leftarrow m'} \quad \forall m < m', \forall i \in [N_p], \forall j \in [l_i], \tag{12}$$

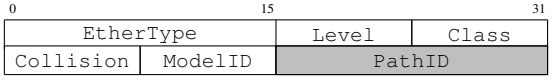


Figure 5: Structure of the DUNE header. For ease of implementation, all fields are padded to 8 bits. The PathID field is grayed to denote that it is only used in emulation environments to facilitate manual routing configurations.

where l_i is the number of switches on the path P_i and s_i^k denotes the k-th switch on the path P_i (*i.e.*, the k-th element of the sequence representing P_i).

The proposed ILP formulation in (10)–(12) can be resolved using a numerical solver; in our implementation, we used the PuLP [51] linear and mixed integer programming modeler.

As a final remark, it is worth noting that, though DUNE's emphasis on path predictability might appear at odds with traffic-steering mechanisms such as equal-cost multipath (ECMP), Fast Re-Route (FRR), or dynamic traffic engineering (TE)[4], our proposed framework can coexist with such mechanisms. In fact, feeding IGP routes as direct inputs to the DUNE planner makes its output immediately compatible with ECMP and FRR, which follow IGP routes for their operation. While this approach may lead to over-provisioning, it is a necessary trade-off for rapid reactivity—much like the additional FIB resources consumed by standby routes. More generally, DUNE's planner stage is designed for integration with centralized traffic controllers, such as a Path Computation Element (PCE), to generate paths that satisfy complex network constraints.

IV. DUNE USER-PLANE OPERATION

Upon completion of the training pipeline in Figure 2, the ML sub-models are coded as P4 programs and deployed in the target programmable user plane, where the distributed inference is performed at line rate. While the general framework we propose can be adapted to different user-plane configurations–potentially combining diverse types of programmable network devices–the implementation of DUNE in this paper focuses on networks of programmable switches. We next present the new DUNE header used to encapsulate information necessary to the distributed inference process across switches in Section IV-A. We then illustrate the user-plane operation in a simple case in Section IV-B before discussing in Section IV-C the complete local processing logic that allows robust functioning of DUNE in real-world settings.

A. DUNE header

A distributed ML inference requires that dedicated state be propagated across switches participating in the process. We realize this by introducing a DUNE protocol header, organized as illustrated in Figure 5 and inserted between the Ethernet and IP headers of packets belonging to flows to be classified.

The Level, Class, Collision, and ModelID fields are used to exchange intermediate results between sub-models

[4]Traffic Engineering (TE) alone refers to the ability to explicitly control traffic paths; it does not inherently imply that such control is applied dynamically, as path configurations may be static or updated infrequently depending on the network design.

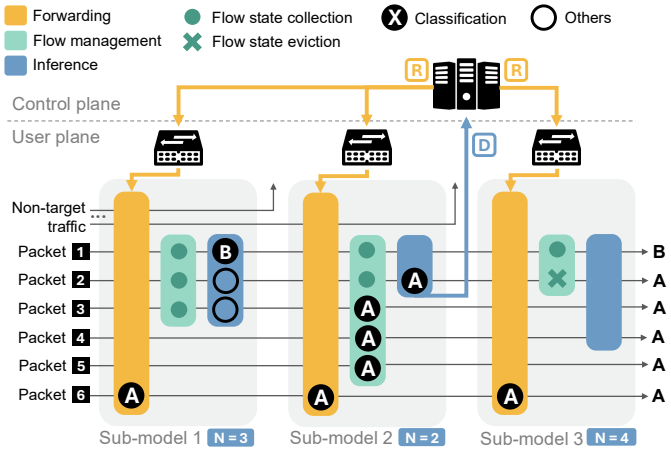


Figure 6: Simple example of the DUNE user plane operation with a distributed ML model over three switches performing inference on packets of one target flow.

and to provide runtime feedback to the control plane. Specifically, the `Level` field indicates whether the classification operates at the flow or packet level, while the `Class` field encodes the corresponding classification outcome. The `Collision` field signals hash collisions by flagging packets whose flow identifiers have collided at any switch along the path. Finally, the `ModelID` field ensures that sub-models are executed in the intended sequence. The `EtherType` and `PathID` fields serve protocol and forwarding functions. The `EtherType` field duplicates the value originally present in the Ethernet header, which is overwritten upon insertion of the DUNE header. The `PathID` field enables MPLS-like forwarding across the network and is useful exclusively in emulated environments–such as the one we present in Section VI–to easily route flows according to manually defined paths.

Inserting an additional header between the Ethernet and IP headers may lead to size constraints when the IP header occupies the entire payload space. This limitation is not unique to our design, as other protocols—such as MPLS—encounter similar challenges. However, the use of jumbo frames mitigates this issue by allowing Ethernet payloads to extend the standard MTU of 1,500 bytes, typically up to 9,000 bytes.

B. Illustrative example of distributed inference

The high-level operation of DUNE in the user plane is illustrated in a simple example in Figure 6, where a sequence of three ML sub-models are deployed into three switches. Each switch is programmed so as to perform three operations on a transiting packet: (*i*) *forwarding* is the baseline function of the switch and consists in redirecting incoming traffic to its next hop or to the ML processing, depending on whether the flow is a target for inference or not; (*ii*) *flow management* is only applied to flows for which inference needs to be run and stores stateful information about the flow-level features and situation (*e.g.*, classified or not) of every target flow; (*iii*) *inference* is the actual Random Forest process that classifies a packet from the available input features. We implement these three operations as in Jewel [5]. Importantly, this means that DUNE employs a single RF model to implement a joint packet- and flow-level approach in the inference component: the first packets of a flow are classified using features extracted from each individual packet only, whereas packets from the N-th leverage additional flow-level features collected from the first N packets of the same flow. Here, N is a configurable parameter that is determined during RF training [5].

Once the network is programmed, the operation is as shown in Figure 6. Traffic that is not a target for inference is normally forwarded by the switches according to their legacy forwarding tables. When the first packet of a flow that has to be classified is received by the entry switch (*e.g.*, packet 1 in the figure, whose flow we assume to belong to class A), it is forwarded to the local flow management stage, where flow-level information is collected and stored, and then moved to the inference stage. Since the first ML sub-model in this example performs flow-level inference on packet N=3, packet 1 of the flow is classified using packet-level features only, and in this specific example, is misclassified as belonging to class B. The packet is then moved along the distributed ML pipeline and traverses the two remaining switches: in both downstream switches, packet 1 is identified as a target for classification; hence, flow-level information is collected, but the packet skips the inference stage since it has already been tagged with a class, although incorrectly (*i.e.*, the `Class` field is set to 'B').

When packet 2 arrives, it follows a similar path. However, let us assume that this time the packet-level inference run by the first switch does not tag packet 2 with a class but marks it as *others*, *i.e.*, belonging to a class pertaining to a downstream ML sub-model as discussed in Section III-E. Hence, packet 2 is forwarded to the second switch (with the `Class` field set to 'others' and the `ModelID` field set to '1'), which updates its internal flow-level state with information from this packet. In the example, the second ML sub-model is, in fact, responsible for inferring packets and flow belonging to class A; also, the flow-level inference point in this model is N=2. The sub-model tags packet 2 as pertaining to class A and updates its local flow management to automatically associate all following packets for the same flow to class A. Packet 2 is forwarded to the third switch, which lets it evict the associated flow-level data from the flow management registers since the flow has now been classified by an upstream switch (*i.e.*, the `Level`, `Class`, and `ModelID` fields in the packet received by the third switch are set to 'flow', 'A', and '2', respectively).

The classification of the flow also triggers a digest message D from the second switch to the control plane; this lets the centralized controller inject new forwarding rules R in all switches and release the flow-level registers. Once the control plane has injected the updated rules, the forwarding stages of all switches tag subsequent packets with their corresponding class and forward them accordingly. This is illustrated, *e.g.*, by packet 6, which is processed after the rule update.

C. Switch local processing logic

The simple example in Section IV-B illustrates the operation of DUNE in a straightforward linear network topology under ideal settings. In practice, the distributed inference process

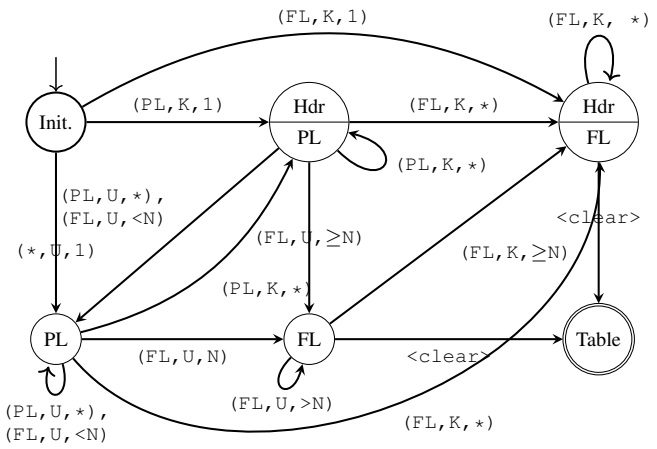


Figure 7: State machine for the local processing logic for one traffic flow at a switch. States indicate the action taken by the switch on incoming packets, whereas edges represent state transitions triggered by the reception of packets with specific DUNE header information.

has to operate in real-world, complex networks where packet reordering and loss can occur, which creates a variety of corner cases whose management requires a careful and robust user-plane implementation.

To ensure that DUNE correctly operates in all possible scenarios that may happen in realistic user planes, we design a state machine that tracks the provenance of classification results within each switch. This state machine governs how incoming packets are interpreted based on the data carried in the packet headers from upstream switches. It thus defines the logical pipeline executed inside the switch to determine whether an inference result needs to be produced locally or if only propagation to downstream switches is instead required.

Figure 7 illustrates the states and their associated transitions. Each state corresponds to an action executed by the switch. State transitions are triggered by three inputs: (i) the classification type, (ii) the result status, and (iii) the flow packet count. Specifically, the classification type is either PL (packet-level) or FL (flow-level), indicating whether the classification was performed at the packet or flow level in upstream switches and extracted from the Level field of the DUNE header. The result status is denoted as U (Unknown) or K (Known), where a result is considered Known if it matches one of the predefined classes in the dataset, and Unknown if it is classified as *others*, as observed in the Class field of the DUNE header. The last input, *i.e.*, the packet count in the flow, is maintained locally by the switch as part of the DUNE flow management routine.

The state machine begins in the Init state, where no classification has yet been performed. Upon receiving an incoming packet, the switch evaluates its classification type, result status, and packet count to determine the appropriate transition. If the first packet of a flow is classified as FL-K in one of the upstream switches, the current switch bypasses inference and simply propagates the result to downstream switches, while also sending a digest to the controller to refresh the memory associated with the flow and to update the forwarding table accordingly. The switch remains in the same state of Hdr/FL for all subsequent packets belonging to the same flow. If the first packet of a flow arrives with PL-K from upstream switches, it transitions to the Hdr/PL state and propagates the packet-level result. It continues to propagate packet-level results—and stay in that state—as long as each new packet is again classified as PL-K by an upstream model. Even if the packet count is equal to the inference point N, the switch does not run FL inference and waits for the FL inference result from the upstream switches, as per the following Assumption IV.1.

Assumption IV.1. *The ML sub-models $m_1 < \cdots < m_n$ are ordered from best to worst according to the model sequencing stage ⑤ described in Section III-F. Hence, for any two models $m_i < m_j$, the output produced by m_i is more accurate than that produced by m_j, when applied to the same input.*

While the switch is in the Hdr/PL state, the arrival of a packet with FL-K (*i.e.*, with a class determined by a flow-level inference in an upstream switch) triggers a system transition to the next state, Hdr/FL, where the switch propagates the FL decision and sends a digest. If instead a packet arrives with a classification of FL-U and the observed packet count is greater than or equal to the inference point, the switch moves from the Hdr/PL state to the FL state, where local flow-level inference is executed. The switch sends a digest to clear its registers and update tables accordingly upon classifying the flow as Known, as in the following Assumption IV.2.

Assumption IV.2. *If the local result is FL-K, we initiate the process to populate tables accordingly and clear registers of the switches on the path of the flow.*

Given a switch in the Hdr/PL state for the target flow, the reception of a packet with PL-U or FL-U and a packet count less than the inference point lets the switch transition to the PL state, where local PL inference runs. A transition to this state also happens when a switch in the Init state receives a packet with any Unknown classification result. So, the switch transitions to this state only if there is an Unknown classification result due to the following Assumption IV.3, which forms the basis for Assertion 1 and Assertion 2 below.

Assumption IV.3. *A flow of packets goes through models following the order provided by model sequencing.*

Assertion 1. *A result of type PL can only be produced locally if it was previously classified as PL-U or FL-U.*

Assertion 2. *A result of type FL can only be produced locally if it was previously classified as FL-U.*

The switch transitions from the PL state to the Hdr/FL state upon receiving a packet with FL-K. If it receives a packet with PL-K, it transitions again to the Hdr/PL state to propagate the packet-level classification results obtained by upstream models. It stays in the same state, PL, where it runs PL inference locally, as long as it receives packets with PL-U (due to Assumption IV.1) or FL-U with a packet count less than the inference point. Whenever the switch receives a packet with FL-U at the inference point, it transitions to the

FL state to classify the flow and propagate the local result to the downstream switches.

Finally, the switch remains in the FL state unless it receives a packet with FL-K. In that case, it proceeds to Hdr/FL to directly propagate the result to the downstream switches.

V. Implementation

We implemented DUNE on both software and hardware P4 targets. The former allows exploring the operation of the framework on complex topologies and at scale, whereas the latter lets us validate its viability with limited topologies but real-world production-grade devices. Due to fundamental differences in the target architectures, the two implementations vary in terms of restrictions and performance, as detailed next.

A. bmv2 software target

Our software implementation is written for the v1model architecture and runs on the simple_switch_grpc target [52], which is based on the bmv2 framework [16]. This software switch offers high flexibility–virtually infinite resources only bounded by the servers where the emulation runs–but low performance, *i.e.*, it is intended for testing, prototyping, and research, but not production scenarios with moderate to high throughput. However, its nature makes it possible and affordable to spawn many software instances and compose large arbitrary topologies. Moreover, given the cost of hardware switches and the recently discontinued production of Tofino ASICs, this software implementation is particularly valuable in making DUNE accessible to the community at large.

Unfortunately, the limited performance of bmv2 also brings functional limitations. For instance, using time-based features for inference in a bmv2 implementation is impractical, since such features depend on high-precision timing when recording packet arrival times, which the software cannot guarantee. The non-deterministic packet processing time of bmv2 can also curb stateful processing, since it introduces variability at runtime with concurrent flows that might, or might not, contend for resources. In addition, bmv2 also has significant shortcomings when it comes to interfacing with control plane agents. On the one hand, its legacy Thrift-based interface does not support the digest mechanism, which is critical to request table insertions. On the other hand, the current gRPC-based interface, while supporting digests, does not support the manipulation of registers, key for evicting classified flows. More challenges come from the library used by bmv2 to interface over gRPC, *i.e.*, *P4Runtime-Shell*, which makes use of global variables and precludes the establishment of parallel connections from the same agent, de facto forcing us to handcraft the gRPC messages instead.

As a workaround to all these issues, our control-plane design is multi-threaded, with each thread dedicated to a single device and holding one gRPC and Thrift connection. This lets us correctly implement a central controller connected to all the devices running DUNE ML sub-models, which, in line with the user-plane operation presented in Section IV, yields two important practical benefits upon classification of a flow: (i) the controller can populate table entries in all devices, significantly reducing the time-to-classification when results are obtained at downstream models; and (ii) the controller can free registers selectively, avoiding the eviction of flows with colliding register indices. This is particularly relevant in multi-path network topologies, where diverse flows mapped to the same register index can have their state stored at different switches–hence a global eviction at all switches based on the index would result in the deletion of erroneous flows.

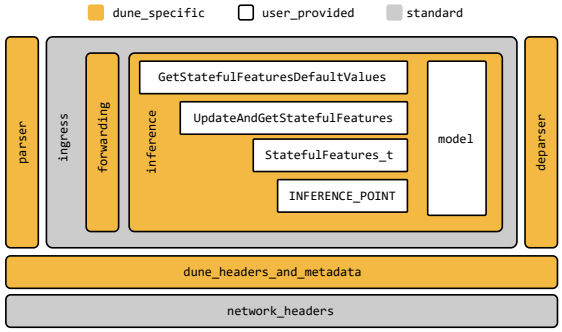


Figure 8: DUNE software implementation blocks, separating elements that are part of the standard bmv2 emulator, specific to our DUNE implementation, or expected to be provided by the end user in order to run a specific inference task.

Table I: Comparison of v1model (bmv2) and TNA (Tofino1) architectures, across five dimensions (rows).

Dimension	v1model	TNA
Pipeline	parser → ingress → egress → deparser →	parser → multi-stage ingress → traffic manager → multi-stage egress → deparser
Resources	Abstract, unlimited	Hardware-constrained (*e.g.*, stages, SRAM/TCAM, ALUs)
Externs	Counters, meters, registers, clone/resubmit	Tables, selectors, multicast, queueing, PRE, hashing
Limitations	non-deterministic performance; timing precision	Stage/table limits; unsupported operations (*e.g.*, unbounded tables, arbitrary recirculation, arbitrary packet modifications)
Use case	Prototyping & education	High-performance deployment

Finally, it is worth remarking that our DUNE implementation is modular and only requires users to input a few model-dependent blocks and variables before running the framework. As illustrated in Figure 8, users shall provide: (1) the StatefulFeatures_t type, encapsulating the set of required stateful features; (2) the UpdateAndGetStatefulFeatures and GetStatefulFeaturesDefaultValues control blocks, encapsulating the update logic and default values for the provided stateful features, respectively; (3) the INFERENCE_POINT constant indicating the number of packets used for feature computation; and, (4) the InferenceModel block encoding the trained RF as a set of feature and code tables [12].

B. Intel Tofino hardware target

Our hardware implementation of DUNE is coded for the Tofino Native Architecture (TNA) and targets the Tofino 1 chipset. Unlike the v1model architecture, which offers concep-

tually infinite resources, the TNA exposes a realistic and constrained pipeline. Table I summarizes the differences between the v1model and TNA across several dimensions. This lets us highlight the most challenging constraints of a hardware implementation: (i) table sizes, stage placement, and match types are fixed; (ii) stateful actions must respect atomicity per stage, limiting concurrency; (iii) arbitrary packet header manipulations or timestamp-based logic are hardware-limited, *e.g.*, divisions are not supported. However, once successfully compiled, TNA-based programs offer strict guarantees on line-rate throughput and deterministic processing time.

Owing to the limitations above, model implementations are case-specific for the Tofino targets, often requiring careful code engineering and refactoring to fit the P4 program in the ASIC. Consequently, providing a user-friendly DUNE framework equivalent to the software one is extremely challenging in the presence of real-world hardware. While we engineered the software implementation to be amicable to hardware–*e.g.*, avoiding unsupported operations such as divisions, ensuring a single per-packet access per register, and using standard-length words at 8, 16, or 32 bits–ensuring that ML sub-models fit the ASIC stages (*e.g.*, 12 in the case of Tofino 1) is a non-trivial task left to the end users.

Specifically, the DUNE implementations we use in our experimental validation employ the efficient Planter mapping of sub-model RFs to the Match-Action Units (MAU) of the Tofino 1 [21]. With careful design, this mapping lets multiple models run on the same switch within a single P4 program, sharing flow management registers and common feature state. The co-location of ML sub-models within the same switch inherently increases the consumption of SRAM and TCAM needed to encode the different RFs; however, it does not necessarily increase the number of consumed stages, since the tables of different sub-models can be evaluated in parallel and hence compiled into the same MAUs. This is an important observation, as stage consumption is the most severe constraint of the target hardware and stays primarily determined by the number of dependent operations in the critical path of the pipeline within each RF.

As far as the control plane is concerned, our DUNE controller binds to the switches using the Barefoot Runtime Interface (BRI) and performs the initial switch setup: activating the ports and loading the trained ML models as table entries in the corresponding P4 tables. During inference, the control plane collects the digest messages that include classification results and, based on such information, updates registers and tables in the user plane at run time. We remark that control-plane-related issues in hardware are also different from those in the software implementation. For instance, the line-rate processing of the Tofino can result in digest message bursts if the replayed traffic is bursty; indeed, such messages are transmitted over the Linux IP stack and, thus, are subject to drops upon congestion, hindering results collection. The alternative of capturing egress traffic is equivalently challenging for the same reason. Thus, we still adopt the digest approach, which, in our experiments, results in a negligible 0.83% of non-tracked flows due to digest loss.

VI. Experimental setup

We test the hardware and software implementations of DUNE in relevant emulation and experimental environments (Section VI-A) reproducing simple linear and real-world fat tree network topologies (Section VI-B). Across these configurations, we deploy DUNE to solve challenging inference tasks (Section VI-C) and evaluate its performance via sensible performance metrics (Section VI-D), comparing it against four state-of-the-art benchmarks (Section VI-E).

A. Emulation and experimental environments

We evaluate the hardware implementation of DUNE in an experimental testbed that comprises 2 servers and 3 programmable switches operating as a comprehensive 100-Gbps platform. The servers are equipped with AMD EPYC 24-core processors running at 2.8 GHz, with 128 GB of RAM, and QSFP28 interfaces. The switches are production-grade Edgecore Wedge 100BF-QS programmable switches, featuring Intel Tofino BFN-T10-032Q chipsets and 32 100GbE QSFP28 ports. All switches run the Open Network Linux (ONL) on top of Debian GNU/Linux 10 (Buster) as the operating system, and use the Intel Software Development Environment (SDE) version 9.13.2 for compiling P4 programs. In the experiments, we distribute the inference process across the switches using DUNE, with one server dedicated to implementing the control plane; the other server is used to inject traffic in the network. The results of inference experiments in the experimental platform are retrieved via digests, which convey the inference outputs to the control plane.

The emulation environment employed to test the software version of DUNE builds on Mininet [18]. Specifically, we employ the `simple_switch` software target for the nodes, which are deployed on a server with 2.8 GHz AMD EPYC 24-core processors and 128 GB of RAM. The setup allows emulating large networks composed of hundreds of switches and injecting real-world traffic into those systems. To collect experimental results in emulation, we process the output PCAP files generated by the `simple_switch` at egress switches. Using `tshark`,we extract both the packet five-tuple and the DUNE header, which carries the classification result and the information about the collision.

B. Network topologies and traffic routing

We consider three network topologies, each selected to isolate and evaluate specific aspects of DUNE's pipeline.

Linear topology. This is a simple network consisting of a chain of switches, where each switch forwards traffic to the next until the destination is reached. Hosts are attached to the edge switches at either end of the chain. In this setup, the host connected to one edge switch acts as the source host, generating traffic, while the host at the opposite end acts as the sink, receiving traffic, ensuring that all traffic traverses the entire chain. In this topology, we replay input traces from the source host to the sink host, where the results are collected; the linear structure ensures that collisions are limited to those originated by simultaneous flows with identical hashes.

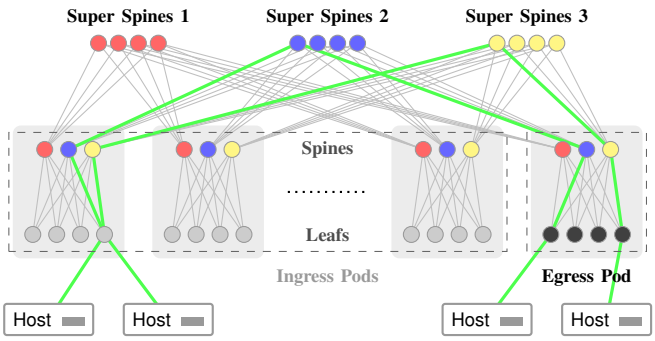


Figure 9: Fat Tree topology considered in our experiments. The green paths exemplify a North-South communication flow from multiple ingress pods to the egress pod.

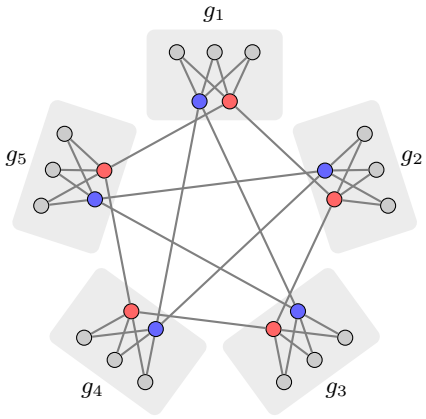


Figure 10: An instance of a Dragonfly+ topology considered in our experiments ($g = 5, a = 2, h = 2, p = 3$). Hosts have been omitted as we focus on the all-to-all traffic pattern.

Fat Tree topology. This topology is a specific instance of the Clos topology, which is widely used in modern data centers for its scalability and high bandwidth. Figure 9 illustrates our reference architecture based on a Fat Tree topology, inspired by large-scale real-world networks. Its design consists of three hierarchical layers: a leaf layer at the bottom, a spine layer in the middle, and a super spine layer at the top. Leaf and spine switches are organized in pods, where leaf switches connect to hosts and forward traffic to spine switches within the same pod. The spine switches, in turn, connect upward to multiple spine planes, each consisting of a group of super spine switches. In the visualization of Figure 9, spines with the same color belong to the same spine plane, emphasizing the logical grouping and parallelism in the topology. This structure enables multiple parallel, equal-cost paths between any pair of ingress and egress pods. In our experiments, we inject traffic through the ingress pods, thereby distributing the load across pods and paths. Traffic is then directed toward hosts connected to the egress pod in a many-to-one pattern. This mimics North-South communication within a data center, where the egress pod provides external connectivity. As shortest paths concentrate near the destination pod, the number of flow collisions increases. This can be anticipated by comparing the traffic-weighted betweenness centrality values of the egress pod edges with those of the ingress pods.

Dragonfly topology. This direct-connect topology is primarily used in high-performance computing (HPC) for its low diameter and cost-effectiveness. Compared with a fat tree topology, it reduces the number of required interconnections, resulting in lower latency while achieving high bandwidth [53]. This is due to a hierarchical structure in which switches are organized into groups with dense local connections and a limited number of global links between groups. While the intra-group and inter-group topologies can be arbitrary, the group hierarchy minimizes the network diameter (typically 3 hops: two local and one global). A dragonfly instance is defined by the number of groups g, the number of switches in each group a, and the number of channels per switch used to connect to other groups h [5]. Figure 10 shows a particular Dragonfly instance with a spine-leaf intra-group topology, known as Dragonfly+ [54]. In this variant, a controls the number of spines, while the additional parameter p controls the number of leafs. Dragonfly's performance across all-to-all, nearest-neighbor, and scatter/gather traffic patterns makes it a popular choice for HPC. In our experiments, we focus on the hardest case for DUNE's planner: all-to-all.

C. Inference use cases

We select two complex classification tasks in device identification and attack detection that are based on publicly available real-world measurement data to show how DUNE performs across different and demanding use cases.

UNSW-IoT [55] is a device identification use case based on traffic measurement data collected from 26 Internet of Things (IoT) and several non-IoT devices. Measurements are conducted in a living lab emulating a smart environment over a period of 6 months. In order to detect which of the 26 devices that traffic flows belong to, we train ML models with 15 days of data and test with one day of data.

ToN-IoT [56] is an attack detection use case where benign, and 9 types of cyberattacks are generated within a representative medium-scale testbed. This testbed comprises several IoT and industrial IoT (IIoT) devices, along with various non-IoT devices. For our analysis, we use 75% of the data for training and the remaining 25% for testing. While the dataset contains benign traffic and nine types of cyberattacks, three of the latter are excluded from our analysis since they have insufficient samples in the test set. This results in the final task of categorizing traffic into seven classes.

D. Inference accuracy metrics

We assess the performance of DUNE and of the selected benchmarks using the F1 score commonly used for evaluating classification accuracy. This metric is calculated based on three key measures: true positives (TP), false positives (FP), and false negatives (FN) as F1=100×2TP/(2TP+FP+FN). We compute the F1 score by averaging the user-plane classification results with three different methods: (i) *micro* average, which sums the TP, FP, and FN across all classes before calculating the metric; (ii) *macro* average, which is the mean of the F1 scores of each class; and (iii) *weighted* average, which weights the F1 scores

[5] Note that $g = a \cdot h + 1$.

of each class by the number of samples in the dataset. For a fair assessment of the performance of the different models, we use a *flow-level* metric that operates on packets but removes the bias caused by long flows [57]: the score is calculated with packet-level TP, FP, and FN where each packet has a weight $w = 1/l$, being l the length of the flow the packet belongs to.

E. Benchmarks

We benchmark DUNE against four state-of-the-art in-switch classification solutions: Mousika [9], Flowrest [13], Netbeacon [32], and Jewel [5]. All use monolithic ML models.

Mousika is a packet-level (PL) classifier leveraging knowledge distillation. In this approach, a Random Forest (RF) or Decision Tree (DT) is trained and distilled into a single Binarized Decision Tree (BDT). We implement Mousika using the publicly available source code [58] and use our model analysis approach to run a grid search over the hyperparameters and PL features, which are later binarized and used to train the teacher RF or DT model. As the BDT student model uses a compact binary-tree representation, we need not limit its size.

Flowrest is a resource-aware, tree-based solution for flow-level (FL) inference. Thanks to its flow management strategy, it allows tracking and classifying flows directly in the data plane. Yet, during the flow feature collection phase, it is unable to classify packets; thus, it is not suitable for classifying traffic with predominantly short-lived flows. The Flowrest instance we use is based on the available source code [59].

Netbeacon is a tree-based solution capable of performing PL classification for short flows and FL classification otherwise. To this end, Netbeacon deploys 3 sets of models: an XGBoost model to predict flow sizes; a PL classifier for short flows, collided flows, and pre-FL-inference packets; and FL classifiers with variable inference points—depending on the flow length—to classify long flows. Depending on the flow-size classification result, FL features are computed and stored—for long flows—or not. Our implementation is based on the publicly available source code [60].

Jewel is a joint packet- and flow-level inference solution. As the previously discussed benchmarks, it is based on tree models. However, unlike NetBeacon, which uses multiple models, Jewel uses a single model for both PL and FL classification. Packets in a new flow are initially classified based on stateless PL features but are used to collect FL statistics. Once a predetermined n-th packet is received, FL features are used for inference. Unlike NetBeacon, Jewel employs a unified RF for both PL and FL inference: during the PL phase, the FL features are assigned constant values, and upon the arrival of the n-th packet, the model switches to FL inference since FL features become available. The FL decision on the n-th packet is then applied to all subsequent packets of the flow. We implement Jewel using the publicly available code [61].

VII. Evaluation

We use the linear topology for two feasibility experiments. First, to demonstrate the practical viability of DUNE on coordinated hardware targets (§VII-A), and second, to validate the consistency between its software and hardware implementations (§VII-B). Given the limited scale of our physical testbed, these experiments provide a necessary baseline to support subsequent evaluations in the emulation environment. Specifically, we rely on emulation to study DUNE at scale along two key operational dimensions (§VII-B): (i) the impact of complex multi-path routing and the resulting flow collisions on distributed inference, and (ii) DUNE's ability to reduce network resource usage by leveraging topological centrality in a representative selection of Dragonfly and Fat Tree topologies with varying sizes and configurations.

Dataset	F1-Score	Mousika	Flowrest	Netbeacon	Jewel	Dune
UNSW	Macro	64.921%	40.100%	46.057%	65.718%	**70.263%**
	Micro	83.568%	41.918%	58.192%	79.132%	**84.291%**
	Weighted	**83.96%**	40.614%	59.278%	79.776%	83.296%
ToN-IoT	Macro	47.099%	55.367%	47.468%	61.718%	**67.388%**
	Micro	42.335%	62,038%	42.380%	67.045%	**72.803%**
	Weighted	37.826%	61.723%	38.024%	65.153%	**72.392%**

Table II: Comparative evaluation of inference accuracy.

		Benchmarks				Dune		
Dataset	Resource	Mousika	Flowrest	Netbeacon	Jewel	SW1	SW2	SW3
UNSW	TCAM	27.80%	14.90%	36.10%	15.60%	9.40%	8.70%	8.70%
	SRAM	0.70%	10.10%	13.50%	12.40%	17.00%	13.00%	16.10%
ToN-IoT	TCAM	1.40%	14.20%	20.50%	5.60%	2.40%	5.60%	8.00%
	SRAM	5.50%	14.70%	19.50%	15.10%	12.70%	16.40%	15.90%

Table III: TCAM and SRAM usage across all models.

A. Small-scale comparative evaluation in hardware

We first test the hardware implementation of DUNE in our experimental platform with a simple linear topology of three Intel Tofino switches as presented in Section VI. The aim of this part of our evaluation is twofold: first, it lets us prove that DUNE can be deployed in real-world environments, by verifying its line-rate operation and efficient switch resource usage; second, it allows us to compare the performance of distributed inference against that of traditional monolithic models proposed in the literature in dependable settings.

Accuracy. We begin by comparing the performance of distributed inference executed by DUNE with the benchmarks performing monolithic inference across three different metrics. In order to assess the accuracy of the solutions, we adopt different flavors of the F1 score as detailed in Section VI-D.

Table II shows how DUNE consistently outperforms all monolithic solutions across all metrics in both use cases, with the exception of the Weighted F1 score in the UNSW dataset, where the PL solution Mousika performs slightly better. The performance improvement of DUNE, compared to the second-best model in both use cases, ranges from 0.7% to 7.5% in all metrics, with an average gain of 5.2% in the Macro F1 score and an advantage of 20% or more over the worst benchmark.

Figure 11 breaks down the F1 score on a per-class basis for each solution. Mousika and NetBeacon exhibit a significant imbalance across classes, whereas NetBeacon and Jewel have more balanced performance, yet still encounter challenges in accurately classifying certain classes compared to others. DUNE, on the other hand, shows more balanced accuracy than other solutions, except for the class representing benign traffic, which is best classified by all solutions. DUNE achieves a

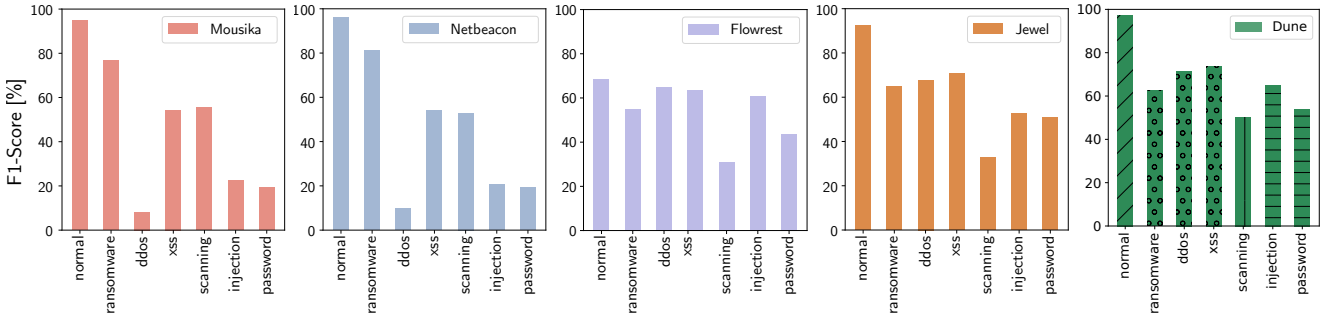


Figure 11: Per-class F1-score obtained by DUNE and the benchmarks in ToN-IoT. The hatch in DUNE indicates the sub-model targeting each class.

Dataset	No Inference	Mousika	Flowrest	Netbeacon	Jewel	Dune
UNSW	852.54	871.64	966.72	989.67	981.47	1137,70
ToN-IoT	852.54	869.18	1001.97	1004.43	1015.08	1183,61

Table IV: Latency in *ns* of all models across three switches.

more homogeneous accuracy across all classes by breaking down the main inference task into dedicated sub-models, as represented with different hatches, ensuring that each class receives adequate attention.

Resource usage. Due to the constraints of programmable network hardware, evaluating resource usage is essential. Table III shows the SRAM and TCAM resource usage of all the benchmarks and DUNE in both use cases. We show the resource consumption of all the switches in which the sub-models are deployed. In UNSW, we implement two sub-models per switch, while in ToN-IoT, we implement two models in the last switch and one in the other two. In terms of SRAM consumption, Mousika is the most efficient as it does not require flow-level management. There is no general gain or loss in the consumption across switches hosting distributed models compared to FL and hybrid solutions. However, given the fact that multiple models are typically deployed per switch, the SRAM consumption of individual sub-models is lower than that of monolithic models, except for Mousika. The reason is that we utilize only one register if multiple sub-models use the same FL features in each switch. Regarding TCAM, the scarcest resource in programmable switches, DUNE shows, in UNSW, lower consumption across all switches compared to other benchmarks, despite deploying two sub-models per switch. In ToN-IoT, besides Mousika, which consumes less but brings a lower score, the TCAM usage in the switches running a single model is lower than the monolithic solutions.

Latency. Finally, we calculate the end-to-end latency of the benchmarks and DUNE in all use cases across three switches, in Table VII-A. The *no inference* case is a baseline where packets are simply forwarded by the switches. For monolithic models, packets passing through the first switch incur both forwarding and inference latency, while packets traversing subsequent switches are subject only to forwarding latency. In DUNE, all switches combine the latency of forwarding and inference. The results show how DUNE realizes line-rate inference with a negligible added delay of around 100 ns per switch with respect to a pure forwarding operation. This delay is on par with that of solutions with a flow-management functionality. While distributing inference logic across switches in DUNE introduces additional end-to-end latency, the impact remains negligible when compared to typical end-to-end delays. Indeed, the inference latency added per sub-model is strictly limited, and the planner, described in Section III-G, strategically minimizes the number of switches required to host the sub-models.

Table V: F1 score comparison for the ToN-IoT use case when DUNE is deployed in a 4-node linear topology and a 36-node Fat Tree topology, both emulated in software.

Class	Linear	Fat Tree	Deviation
Normal	96.7	97.0	0.3
Ransomware	53.5	58.3	4.8
DDoS	67.7	66.2	1.5
XSS	72.8	72.3	0.5
Scanning	48.8	48.1	0.7
Injection	63.5	60.5	3.0
Password	58.2	54.9	3.3
Macro	65.8	65.3	0.5
Micro	71.5	70.4	1.1
Weighted	71.1	69.8	0.3

B. *Large-scale scalability assessment via emulation*

We now consider the software implementation of DUNE and employ the emulation environment to run additional experiments. The aim of these tests is twofold: first, they allow proving the credibility of the software version of DUNE by juxtaposing its performance in a `mininet` topology against that of the hardware implementation running on an equivalent network of real-world switches; second, they let us explore the scalability of DUNE to large topologies with complex routing. The evaluation is here limited to the ToN-IoT use case since, unlike the UNSW-IoT task, its inference process does not rely on time-based features that, as explained in Section V-A, are not properly handled by the bmv2 model. We split the input PCAP trace into as many subsets as there are hosts, assigning one subset to each ingress host. The splitting is performed at the flow level, ensuring that the number of flows is evenly distributed across hosts[6].

Validation of the software implementation. Figure 12 compares the accuracy of the two versions of DUNE, *i.e.*, hardware and software, once deployed in a linear topology. Such a topology is realized with 3 real-world Intel Tofino switches for the hardware version of DUNE and with 4 switches emulated in `mininet` for the software version.

The various F1 scores are also placed side by side with those obtained by the unconstrained ML model trained in

[6]Note that this traffic assignment does not guarantee traffic balance across hosts, as flows might have varying throughput and duration.

Table VI: Results of DUNE ML sub-model planning for distributing the ToN-IoT models in different topologies. For the Fat Tree, the communication pattern is many-to-one, with each ingress leaf communicating with one egress leaf. For the Dragonfly, the communication pattern is all-to-all. For each configuration, we report the longest model chain (k) that the topology and traffic pattern allow, and the model chain length (n^*) produced by Alg. 1 under this constraint. The 'Nodes' column shows model deployments out of total nodes, and the '%' column displays that ratio.

	DC Size	Pods	Leafs	Spines	Super spines	n*/k	Paths	Nodes	%	Notes
Fat Tree (many-to-one)	XS	2	2	2	1	4/5	4	8/10	80	Tiny DC: testing, labs
	S	4	4	3	1	4/5	12	19/31	61	Small DC: edge, small enterprise
	M	8	6	4	2	4/5	192	46/88	52	Medium DC: medium enterprise, small offsite
	L	16	8	5	2	4/5	960	98/218	45	Large DC: large enterprise, medium offsite
	XL	32	10	7	4	4/5	3100	262/572	45	Very large DC: hyperscalers
	Flavor	g	a	h	p	n*/k	Paths	Nodes	%	Notes
Dragonfly (any-to-any)	Default	5	2	2	-	1/1	118	6/10	60	Tiny DC: testing, labs
	Default	7	3	2	-	1/1	600	14/21	67	Small DC: edge, small enterprise
	Default	10	3	3	-	1/1	1482	20/30	67	Medium DC: medium enterprise, small offsite
	+	*	*	*	*	2/2	*	*/*	100	*

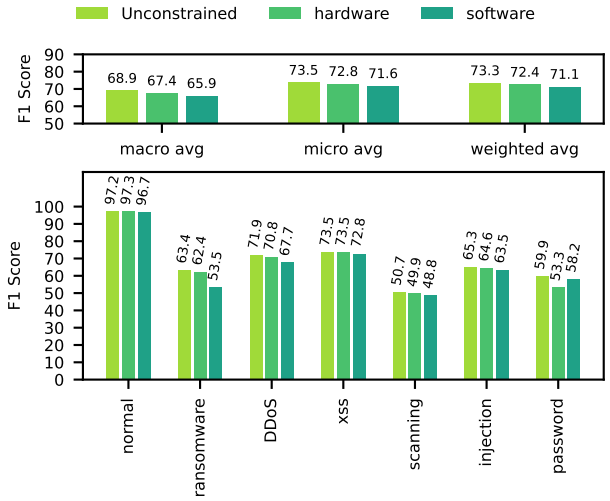


Figure 12: Performance comparison between DUNE software- and hardware-based implementations in the ToN-IoT use case. The unconstrained results are for the best possible ML model running on Python in a high-end server.

stage ① of the DUNE pipeline in Section III, which runs on a high-end server using Python libraries and provides an upper bound to the inference performance. We observe that the hardware and software implementations show very close accuracy, with a discrepancy of 0.1 percent points in the averaged F1 scores in favor of the hardware version (top plot); even when looking at class-level F1 scores, we see differences between 0 and 9 percent points (bottom plot). These relatively small differences are mainly caused by the high latency of bmv2 cross-plane communications discussed in Section V-A, which causes delay and losses in digests, hence additional flow collisions in the switches. Moreover, both implementations of DUNE have performance close to the best case, represented by the unconstrained ML model running in Python, which is unaffected by the complexities of user-plane operations.

Operation in complex topologies We now assess DUNE's capability to operate under the challenging conditions imposed by large-scale, real-world topologies. While Dragonfly topologies are known to exhibit high structural centrality due to their limited number of global inter-group links, this characteristic can change under asymmetric traffic patterns. In particular, under many-to-one communication, Fat Tree topologies can exhibit higher traffic-weighted edge centrality than Dragonfly under all-to-all traffic. Motivated by this observation, we evaluate DUNE's classification performance in a Fat Tree topology, where multi-path routing increases the likelihood of flow collisions, thereby stressing the system.

Our Fat Tree instance consists of 16 leaf switches and 16 spine switches, organized into 4 pods and interconnected through 4 super-spine nodes. We inject traffic flows derived from the ToN-IoT use case at the leaf nodes, as described in Section VI-B, and execute the distributed inference task across the entire network so that every flow is classified by DUNE. Table V compares the resulting F1 scores to those recorded in a simple four-switch linear topology: deviations are small, in the [0.3–1.1] range when considering average metrics and in the [0.3–4.8] range for individual classes. As discussed in Section VI-B, the differences are primarily caused by higher flow collisions in the Fat Tree, yet we deem them acceptable considering the much larger size and higher complexity of the target network.

Exploiting topological structure for resource savings. Finally, we assess the efficiency of DUNE in distributing the inference process while minimizing network resource usage. Because Dragonfly has a lower diameter than Fat Tree, using both allows us to compare the planner's efficiency under both favorable and challenging conditions. To this end, we consider the sequenced ML sub-models returned by stage ⑤ of the pipeline in Section III under the k constraint that each topology and routing mandates; we then run the planning stage ⑥ to deploy such models for a variety of topology parameter combinations.

Table VI reports the number of node deployments required to process the ToN-IoT workload in topologies representative of typical data center networks, from private edge facilities to hyperscalers. In the Fat Tree topology under a many-to-one traffic pattern $k = 5$, Algorithm 1 finds $n^* = 4$. Given this 1-node slack, models can be pushed towards the egress pod, freeing the ingress leaf layer. As a result, the super-spine and egress pod switches are highly central, allowing the topology to scale faster than the number of required models,

thereby resulting in higher efficiency as the network size grows. Meanwhile, in the Dragonfly configurations, the any-to-any traffic pattern imposes $k = 1$ in the default flavor, and $k = 2$ in the '+' flavor. Under such strong constraints, DUNE's planner still finds feasible, efficient solutions in the former case and the trivial solution (deploy everywhere) in the latter. Ultimately, the results demonstrate the effectiveness of DUNE in limiting the resources used by the inference process, both locally on each switch and, when possible, globally across the target network infrastructure.

VIII. Conclusions

We propose DUNE, the first framework that automatically decomposes unconstrained user-plane classification models into simpler sub-models and determines their optimal placement across programmable switches, enabling deployment in network topologies of varying complexity. Results demonstrate that DUNE is topology-aware and can adapt to arbitrary network topologies of varying sizes. Extensive experiments show that DUNE yields higher accuracy than traditional single-device approaches, with lower resource usage and comparable delay.

Our work also opens interesting directions for future research, including different optimization objectives for the partitioning and sequencing problems, the utilization of recent Deep Learning-based architectures for the implementation of the sub-models, or further evaluations with diverse and larger network topologies. To foster these investigations, we have provided public access to DUNE's code at [17].

References

[1] F. Hauser et al. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *ArXiv*, abs/2101.10632, 2021.

[2] E. F. Kfoury et al. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[3] S. Kaur et al. A review on p4-programmable data planes: Architecture, research efforts, and future directions. *Computer Communications*, 170:109–129, 2021.

[4] C. Zheng et al. In-network machine learning using programmable network devices: A survey. *IEEE Communications Surveys & Tutorials*, 26(2):1171–1200, 2024.

[5] A. T.-J. Akem et al. Jewel: Resource-efficient joint packet and flow level inference in programmable switches. In *IEEE INFOCOM'24*, 2024.

[6] C. Busse-Grawitz et al. pForest: In-network inference with random forests. *CoRR*, abs/1909.05680, 2019.

[7] T. Swamy et al. Taurus: A data plane architecture for per-packet ml. *ASPLOS*, 2022.

[8] G. Siracusano et al. Re-architecting traffic analysis with neural network interface cards. In *NSDI*, Renton, WA, April 2022. USENIX.

[9] G. Xie et al. Empowering in-network classification in programmable switches by binary decision tree and knowledge distillation. *IEEE/ACM Trans. Netw.*, 2023.

[10] C. Zheng et al. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *ACM SIGCOMM Computer Communication Review*, 2024.

[11] J. Yan et al. Brain-on-Switch: Towards advanced intelligent network data plane via NN-Driven traffic analysis at Line-Speed. In *21st NSDI*, pp. 419–440, Santa Clara, CA, April 2024. USENIX Association.

[12] Z. Xiong and N. Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pp. 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[13] A. T.-J. Akem et al. Flowrest: Practical flow-level inference in programmable switches with random forests. In *IEEE INFOCOM'23*, 2023.

[14] B. Bütün et al. Dune: Distributed inference in the user plane. In *IEEE INFOCOM 2025*.

[15] C. Zheng et al. Dinc: Toward distributed in-network computing. *Proc. ACM Netw.*, 1(CoNEXT3), nov 2023.

[16] P4.org. Bmv2. https://github.com/p4lang/behavioral-model.

[17] B. Bütün et al. Dune. https://github.com/nds-group/Dune.

[18] M. Team. Mininet: An instant virtual network on your laptop. 2017.

[19] A. Sapio et al. In-network computation is a dumb idea whose time has come. In *HotNets'17*, NY, USA, 2017. ACM.

[20] D. Sanvito et al. Can the network be the ai accelerator? *NetCompute '18*, 2018.

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

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

[23] B. M. Xavier et al. Programmable switches for in-networking classification. In *IEEE INFOCOM 2021*, pp. 1–10.

[24] G. Xie et al. Soter: Deep learning enhanced in-network attack detection based on programmable switches. In *SRDS*, 2022.

[25] H. Siddique et al. Towards network-accelerated ML-based distributed computer vision systems. In *IEEE ICPADS*, pp. 122–129, 2021.

[26] K. Friday et al. A learning methodology for line-rate ransomware mitigation with p4 switches. In *Network and System Security*, pp. 120–139. Springer Nature Switzerland, 2022.

[27] X. Zhang et al. pHeavy: Predicting heavy flows in the programmable data plane. *IEEE Transactions on Network and Service Management*, 18(4):4353–4364, 2021.

[28] K. Friday et al. INC: In-network classification of botnet propagation at line rate. In *Computer Security – ESORICS 2022*, pp. 551–569. Springer International Publishing, 2022.

[29] Z. Zhang et al. Sentinelx: A lightweight malicious traffic detection system based on programmable switches. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*, pp. 1–10, 2025.

[30] S. U. Jafri et al. Leo: Online ML-based traffic classification at Multi-Terabit line rate. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 1573–1591, Santa Clara, CA, April 2024. USENIX Association.

[31] A. T.-J. Akem et al. Henna: Hierarchical machine learning inference in programmable switches. In *NativeNI 22*, pp. 1–7. ACM, 2022.

[32] G. Zhou et al. An efficient design of intelligent network data plane. In *USENIX symposium on security*, 2023.

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

[34] Z. Zhao et al. Rids: Towards advanced ids via rnn model and programmable switches co-designed approaches. In *IEEE INFOCOM 2024*, pp. 1–10, 2024.

[35] M. Zhang et al. Quark: Implementing convolutional neural networks entirely on programmable data plane, 2025.

[36] A. Mestres et al. Knowledge-defined networking. *SIGCOMM Comput. Commun. Rev.*, 47(3):2–10, sep 2017.

[37] A. Sapio et al. Scaling distributed machine learning with In-Network aggregation. In *18th NSDI*, pp. 785–808. USENIX, April 2021.

[38] M. Seufert et al. Marina: Realizing ml-driven real-time network traffic monitoring at terabit scale. *IEEE Transactions on Network and Service Management*, 21(3):2773–2790, 2024.

[39] L. Bracciale et al. The case for native multi-node in-network machine learning. In *NativeNI 22*, NativeNi '22, pp. 8–13, New York, NY, USA, 2022. Association for Computing Machinery.

[40] K. Razavi et al. Netnn: Neural intrusion detection system in programmable networks. In *IEEE ISCC*, 2024.

[41] K. Zhang et al. Design, implementation, and deployment of multi-task neural networks in programmable data-planes. *IEEE Transactions on Network and Service Management*, 23:740–755, 2026.

[42] H. Kim et al. Experience-driven research on programmable networks. *SIGCOMM Comput. Commun. Rev.*, 51(1):10–17, 2021.

[43] N. Sultana et al. Flightplan: Dataplane disaggregation and placement for p4 programs. pp. 571–592.

[44] E. Cramer and G. Prevedello. Per-Class Feature Importance, 2020.

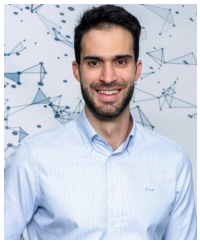
[45] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon et al., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[46] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[47] S. M. Lundberg et al. Consistent individualized feature attribution for tree ensembles. *CoRR*, abs/1802.03888, 2018.

[48] Y. Zhang et al. Pegasus: A universal framework for scalable deep learning inference on the dataplane. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pp. 692–706, New York, NY, USA, 2025. Association for Computing Machinery.
[49] H. Yan et al. Linc: Enabling low-resource in-network classification and incremental model update. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pp. 1–12, 2024.
[50] O. Karrakchou et al. Collaborative p4-sdn ddos detection and mitigation with early-exit neural networks. In *GLOBECOM 2025 - 2025 IEEE Global Communications Conference*, pp. 6081–6086, 2025.
[51] S. Mitchell et al. Pulp : A linear programming toolkit for python. 2011.
[52] P4.org. Bmv2. https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.
[53] J. Kim et al. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.
[54] A. Shpiner et al. Dragonfly+: Low cost topology for scaling datacenters. In *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pp. 1–8, 2017.
[55] A. Sivanathan et al. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8), 2019.
[56] A. Alsaedi et al. TON_IoT telemetry dataset: A new generation dataset of IoT and IIoT for data-driven intrusion detection systems, 2020.
[57] M. Gucciardo et al. Evaluating the impact of flow length on the performance of in-switch inference solutions. In *CNERT 24*. IEEE INFOCOM 2024.
[58] G. Xie et al. Mousika. https://github.com/xgr19/Mousika.
[59] A.T-J. Akem et al. Flowrest. https://github.com/nds-group/Flowrest.
[60] G. Zhou et al. Netbeacon. https://github.com/IDP-code/NetBeacon.
[61] A.T-J. Akem et al. Jewel. https://github.com/nds-group/Jewel.

Beyza Bütün (Student Member, IEEE) is a Ph.D. candidate and Research Assistant at IMDEA Networks Institute and the Department of Telematics Engineering at Universidad Carlos III de Madrid. She earned her B.Sc. (2020) and M.Sc. (2022) in Computer Engineering from Middle East Technical University (METU), where her master's research focused on the optimal design of wireless data center networks. Her current research interests include machine learning in programmable networks, distributed in-band inference, data-plane energy modeling, and efficient distributed AI systems. Her work on distributed in-band inference received the IEEE INFOCOM 2025 Best Paper Award.



David de Andres Hernandez (Member, IEEE) is a Ph.D. candidate and research assistant at IMDEA Networks Institute and Universidad Carlos III de Madrid, Spain. He received the B.Eng. degree in Telecommunications from Universidad Politécnica de Madrid, Spain, in 2018, and the M.Sc. degree in Electrical Engineering and Information Technologies from TU Munich, Germany, in January 2024. As part of his master's thesis, he worked on latency-aware path optimization at IXP route servers at DE-CIX, Germany. He has also worked as a Systems Engineer for Juniper Networks between 2019 and 2021. His research interests include scalable, high-performance packet processing systems, stateful network functions, in-network inference, and inter-domain routing.

Alexis Carré is currently pursuing the M.Sc. degree in Computer Science at École Normale Supérieure de Lyon, France, where he received the B.Sc. degree in 2023. He visited IMDEA Networks Institute, Spain, as a Research Intern in 2025. His research interests include modeling and analyzing the behavior of computer networks, programming languages, compilers, and program transformations.

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 and CTO at Net AI Tech Ltd. He received M.Sc. degrees from University of Illinois at Chicago, IL, USA (2003), and Politecnico di Torino, Italy (2004), a Ph.D. degree from Politecnico di Torino (2008), Italy, and a Habilitation a Diriger des Recherches (HDR) from Universite de Lyon, France (2014). He held tenured positions as Maitre de Conferences (Associate Professor) at Institut National des Sciences Appliquees (INSA) de Lyon, France (2009-2013), and Researcher at Consiglio Nazionale delle Ricerche (CNR), Italy (2013-2019). He has been a visiting researcher at Rice University, TX, USA (2006-2007), Universitat Politecnica de Catalunya (UPC), Spain (2008), and University College London (UCL), UK (2016-2018). He leads the Networks Data Science group at IMDEA Networks Institute, focused on research at the interface of computer networks, data analysis and machine learning.