In-network real-time flow classification using hierarchical decision trees

Teodor Karkashina
School of Computing Science
University of Glasgow
Scotland, UK
t.karkashina.1@research.gla.ac.uk

Awais Aziz Shah
School of Computing Science
University of Glasgow
Scotland, UK
awaisaziz.shah@glasgow.ac.uk

Dimitrios P. Pezaros
School of Computing Science
University of Glasgow
Scotland, UK
dimitrios.pezaros@glasgow.ac.uk

***Abstract*—In-network computing has emerged as a promising approach to offload ML-related functionality from servers to network devices, leveraging the advanced capabilities of programmable network devices and expressive data plane programming languages such as P4. By implementing ML models in the data plane, we can achieve high throughput and low latency inference, reducing server loads, and enhancing response times.**

This work proposes a novel hierarchical deployment for Decision Tree models (HDT) within the network data plane, designed for real-time flow classification of the network traffic. The hierarchical structure breaks down a complex classification into multiple levels each one refining the classification progressively.

Our results demonstrate the feasibility and efficiency of executing Hierarchical Decision Tree models in the network data plane, achieving high performance in classifying a 12-class classification scenario. The HDT performs better in the data plane by 26% compared to a flat Decision Tree implementation and 12% compared to a flat Random Forest implementation.

***Index Terms*—Programmable switch, machine learning, in-network inference, P4**

I. Introduction

The SDN paradigm and data plane programmability [1] (e.g., programmable switches, smartNICs) give the capability to offload or accelerate data processing. SDN supports context-aware, and user-specific services by introducing virtualized network services at the edge of the network, near the end users to reduce end-to-end latency, time-to-response, and unnecessary utilization of the core network, while providing flexibility for resource allocation [4], [24].

Meanwhile, the high throughput and low latency potential of programmable network devices like P4 switches make them suitable candidates for reducing the processing load on edge devices. This opens new opportunities in the context of machine learning, where in-network computing can significantly enhance the performance of ML inference tasks. By deploying the ML models in the network data plane, real-time data processing is achievable, limiting the required communication with centralized servers and reducing latency. Furthermore, the benefits of in-network processing can be applied to multiple applications like key-value caching [21], lock management [22], and DDoS attacks [23].

Performing inference of ML models for real-time classification tasks in programmable switches has various challenges. While programmable network devices offer enhanced processing capability compared to fixed silicon devices, they still impose constraints on the available resources (e.g., limited available memory), the lack of floating-point operations, and the restricted set of ALU operations per pipeline. Therefore, deploying complex ML inference tasks in the data plane with significant performance benefits remains challenging.

Embedding an ML model within a single programmable network device is not trivial since the resource limitations in the programmable network devices limit the size of the ML model that can be deployed [17], affecting the model's performance. Switches use most of their resources for fundamental networking operations but leave available resources for inference tasks [16]. The number of supported Match Action Units (MAUs) is between 12 and 20 per pipeline, with four pipelines per switch, limiting the supported functionality. The memory capacity of the tables is in the range of hundreds of megabits, typically distributed across various pipelines within the programmable network device.

Studies have demonstrated the potential of integrating trained ML models into resource-constrained programmable switches [2], [3]. However, existing solutions often face significant restrictions, especially when coping with complex inference tasks. The increased complexity of the ML models and the large number of classes challenge the existing frameworks to meet performance requirements. Solutions should demonstrate better scalability to complex ML models, especially in multiclassification scenarios, indistinct feature boundaries for classification decisions, and high resource utilization. Deploying very large and complex ML models, such as DNNs, in programmable switches will exhaust the available resources and fail to meet performance requirements [7].

By splitting the complex classification tasks into simpler ones, we can address the issues mentioned above. In this context, this work proposes a tree-like hierarchical structure of the Decision Tree model for traffic flow classification, where higher levels make initial distinctions, with subsequent levels refining the classification to fewer classes until the final classification decision is made. The HTD classifies more classes with better precision compared to Random Forest and Decision Tree implementations. The critical elements of the hierarchical tree classifier include class grouping, hierarchical class relationships, and progressive classification refinement through the multiple levels design. At every level, one or more DT models are executed to decide on the next tree branch. By developing the hierarchical Decision Tree classier, we make the following contributions:

- Enhance the performance of Decision Tree inference for multiclass classification scenarios with a comparative analysis against flat implementations of Decision Trees and Random Forests in the data plane.
- Enable the decomposition of complex classification problems into simpler tasks, reducing class overlap and misclassification.
- Achieve efficient resource utilization and implementation of the algorithm in the ingress pipeline of the P4 v1model architecture.

The remainder of this paper is structured as follows. Section II discusses the literature related to Machine Learning inference offloading approaches. Section III introduces the Hierarchical Decision Tree model and methodology. Section IV discusses the experimental setup and the results. Finally, section V concludes this paper.

II. Related Work

Selecting the suitable ML model for offloading inference capabilities is essential to mitigating the resource constraints in programmable switches. Studies have explored the potential of offloading Neural Networks (NNs), particularly Binary Neural Networks (BNNs), to programmable devices [6], [7].

Although NNs can achieve higher performance, the resource limitations and restricted mathematical operations available in programmable switches present significant challenges. To overcome these constraints NetNN [19] deploys NN across multiple programmable switches (one neuron per switch) to develop an IDS in the programmable data plane, but NNs are more suitable for data plane devices like smartNICs or FPGA-enhanced switches [8], which offer more resources. However, smartNICs are typically deployed at specific locations in the network, providing line-rate inference only at those points, while FPGA-enhanced switches or distributing the NN neurons across multiple programmable switches introduce considerable additional costs when deployed at scale.

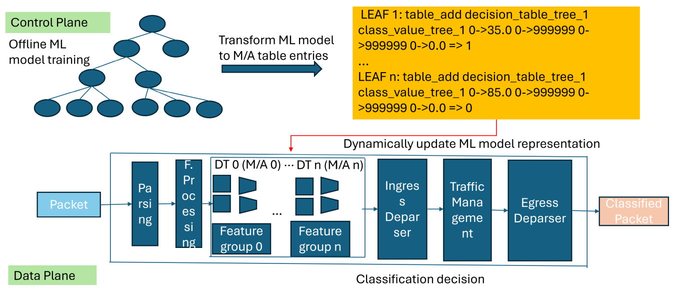
Many works in literature use Decision Trees (DTs) and Random Forests (RFs) for inference on programmable network devices. The DT model is a supervised learning algorithm characterized by a tree-like structure used for solving regression and classification problems. This structure consists of a root node, intermediate nodes, and several leaf nodes, each representing a classification decision. DTs align well with the PISA architecture [5], yielding superior performance compared to other ML models, such as SVM, Naïve-Bayes, and K-means [2].

Two main techniques are used to implement the DT inference into the data plane: the depth-based approach [10], [17] and the encoded-based [3], [9] approach. The depth-based approach maps each level of the DT model to an M/A stage, creating a dependency between the number of stages and the tree's depth. Conversely, the encoded-based approach maps each feature to a M/A table and encodes the ranges of the feature values. The final classification is determined using a code-to-leaf M/A table, which maps the combined feature codes to a specific leaf node. While the encoded-based approach eliminates the correlation between the tree's depth and the M/A stages, it demands more memory as the number of features increases. Ilsy [9] uses programmable switches to map the DT classification model and classify network data. In pHeavy [12], heavy flows are classified while simultaneously the size of the DTs is reduced, and memory is dynamically managed. The pForest [10] solution executes the inference of Random Forest (RF) models within programmable switches to classify traffic flows. The methodology for mapping the RF model inference to the data plane is the same as that of the DT models. Each individual tree within the RF is developed separately, independently classifying the input data, and the final classification decision is realized through a voting table implemented in a M/A table. The voting table applies a majority voting mechanism, aggregating the outcome generated from the individual trees to produce the final classification decision. Planter [3] is a framework designed to translate tree-based ensemble models for deployment in programmable switches. It supports ML models such as Random Forest, XGBoost, and Isolation Forest. It separates the dependence of M/A stages from the tree depth by employing one dedicated table for each tree and one for each feature. The encoded-based model mapping technique used by Planter is also adopted by works like Flowrest [18] and Henna [15], which develop RF flow-level classification and packet-level classification, respectively, for multiclass scenarios in hardware programmable switches.

III. Hierarchical Decision Tree Design

A. Control Plane ML Model Training

The design of our HDT comprises control and data plane components, as shown in Figure 1. The ML model is trained in the control plane and relies on Scikit-Learn libraries for model preparation. Each level of the HDT model is trained separately, using different features at each level and ensuring that the samples processed at one level are either classified or passed down to the next level for further processing. Each level is independent, excluding samples from previous levels or those belonging to different nodes. The optimization of the ML



```
LEAF 1: table_add decision_table_tree_1
class_value_tree_1 0->35.0 0->999999 0-
>999999 0->0.0 => 1
...
LEAF n: table_add decision_table_tree_1
class_value_tree_1 0->85.0 0->999999 0-
>999999 0->0.0 => 0
```

Fig. 1. Overall high-level architecture of HDT showing both the control and data plane.

model at each level is achieved by tuning the hyperparameters using grid search techniques.

The hyperparameters that have the biggest impact on each Decision Tree of the HDT design are max_depth, min_samples_leaf, min_samples_split, and criterion. The min_samples_leaf ensures that there is a minimum number of samples at each leaf node, preventing overfitting. Similarly, min_samples_split specifies the minimum number of samples required to split an internal node, ensuring that splits are done only when there is sufficient data to justify them. Furthermore, the criterion parameter, which is set to gini, measures the impurity of a node and creates balanced splits, making the tree more robust to changes in the data and mitigating the overfitting risk. The max_depth parameter controls the maximum depth of the tree. Deeper trees capture more details from the training data, which can lead to higher accuracy but risks overfitting to the noise of the data. In our HDT, the max_depth parameter varies in each decision tree from 5 to 25. The first level achieves better accuracy with a tree depth of 25, but the rest of the levels use smaller tree depths. The depth of the tree impacts memory usage, and very deep trees are impractical [10], [20]. By using the standard hyperparameter tuning techniques and due to the hierarchical structure, which splits the classification task into smaller sub-tasks at each level, the HDT reduces the configuration rules needed to express the model by 40% compared to a Random Forest implementation with three trees, indicating that HDT uses less memory than an RF implementation. HDT also has the same number of rules as a Decision Tree implementation with a max depth of 15 despite using 5 DT to make the final decision. In a single DT, when the problem is complex and the tree depth is increasing, the number of rules increases exponentially [11], while in HDT, this is not the case due to its hierarchical structure. The feature selection process for the HDT model considers the constraints of the data plane, such as the absence of floating-point and mathematical operations. To circumvent these limitations posed by the programmable switch architecture and the need to bound execution time, we approximate the feature values and represent them as integers. The loss of accuracy after approximating the feature values is between 0.6% and 1.6%, with the flow duration feature value experiencing the biggest loss (1.6%), but this low accuracy loss does not affect the classification result.

The HDT performs flow-level classification, meaning that it keeps track of the state of the feature values per flow over time. To achieve this, it needs to maintain state information. The importance of the features is evaluated by the Mean Decrease in Impurity (MDI) metric, which is used to select the most suitable features for the classification process. These features include flow-level features, such as flow duration, number of bytes transferred from source to destination, number of bytes transferred from destination to source, TCP connection setup time, as well as packet-level features like service type and transport protocol. Figure 2 illustrates the HDT classifying incoming traffic through a series of Decision Trees. The first level separates incoming traffic into 'Normal' or 'Attack'. The 'Normal' flows are forwarded to the appropriate output port, but for traffic classified as 'Attack', the corresponding flows are further classified to Attack groups 1 to 3. The last level Decision Trees classify the traffic to the specific attack type.

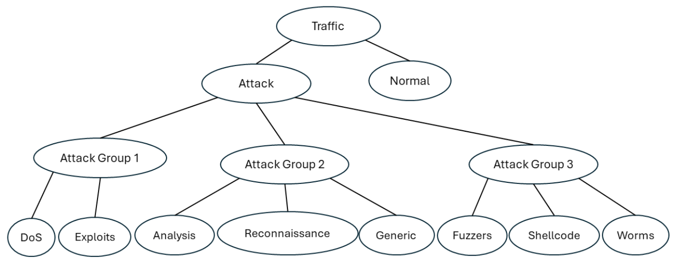


Fig. 2. Hierarchical Decision Tree flow classification of different attack types.

B. Data Plane Inference

The inference task is offloaded to the Data Plane, where the incoming packets are parsed to extract the necessary values from the header. In contrast to similar frameworks like Henna, the proposed hierarchical DT operates solely within the ingress pipeline. It leverages only DT models to classify flows and implements a streamlined model mapping method, using only one M/A table per decision tree. Unlike most existing frameworks [3], [9], [18] in the literature, which use one M/A table per feature and an encoded table, the HDT follows a different direction, which is the simplified encoded-based approach, like Mousika [13], Netpixel [11], and pHeavy [12]. This approach involves a single decision table for classification where the feature values serve as inputs and range-match is used for labelling. This approach can potentially exceed the available memory on a programmable network device if a tree model is complex. By employing HDT, this risk is mitigated since its hierarchical structure necessitates fewer configuration rules to represent the ML model through successive levels. This design saves resources, maintains robust classification performance, and is more suitable for offloading tree-based ML models in programmable networks. Following the simplified encoded approach, we use less memory and fewer stages to map the ML model to the PISA architecture. Moreover, it allows the HDT to scale up in multiclass classification scenarios where an HDT with many levels can be used.

C. Flow Table Design

To manage flow-level classification, a unique identifier is created using crc32 hash checksum derived from the 5-tuple, which includes the source and destination IP addresses, source and destination ports, and transport protocol identifier. Also, an index value is generated, using crc32 hash checksum using the same 5-tuple, to point the flow-id to the register and ensure that each flow is correctly associated with its respective data, such as flow-id, packet counters, etc. The primary role of the flow index is to detect hash collisions, which occur when different inputs can produce the same output, leading to potentially incorrect flow-id classification. The restricted size of the registers and the high volumes of flows passing through the switch can lead to frequent hash collisions, which are challenging to manage and absorb. Collisions in HDT are rare because after the flow is classified, it is removed from the table after a defined time interval. The count field holds the number of packets for each flow, and the class field holds the classification result for the specific flow using the packet recirculation feature. In HDT, we perform early packet classification to achieve faster threat detection and mitigation. Based on our experiments with real packet traces containing various attack types, we observed that the fourth packet in every flow typically provides the most reliable attack type detection. Consequently, we make the classification decision after calculating the feature values from the first four packets of the flow. In this work, we do not take any action for the remaining packets of the flow after the flow is classified as malicious, but since the flow table is updated with the classification decision, a straightforward action is to drop the remaining packets of a malicious-classified flow.

TABLE I
FLOW TABLE INCLUDING TWO FLOWS WITH ITS FEATURE VALUES, PACKET COUNTS, AND CLASSIFICATION DECISION

Flow ID	Count	Class	Feature1	···	Feature n
1	5	1	124	···	12
2	0	-	0	···	9999
3	21	2	10	···	120

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

The experimental setup for evaluating the HDT model was developed on the v1model architecture using the BMv2 software switch. This setup was deployed on a host system equipped with a 4-core Intel i7- CPU and 32GB of RAM. The same host was also used to run the baseline models against which the performance of HDT was compared. We have assessed the performance benefits and feasibility of our HDT design and report on the precision, recall, and F1-score of the HDT model against data plane implementations of Random Forest and Decision Tree models. The DT model has been used to deal with traffic classification problems in previous ML offloading scenarios [3], [9], yielding good performance, and it can serve as a baseline to compare it with the HDT model. The DT model cannot scale well in larger models, requiring more resources compared to HDT. The RF model is an ensemble learning method that uses multiple Decision Trees during training and makes a classification decision based on a majority vote from diverse Decision Trees. Like the RF, the HDT uses multiple DTs to get the final decisions, but in a different manner. The HDT manages complexity through hierarchical structuring, while RFs exploit ensemble learning. HDTs scale by dividing decisions into simpler steps at each level, while RFs scale by adding to the number of DTs. Moreover, HDTs follow a sequential decision process through a hierarchy, whereas RFs make decisions in parallel across multiple decision trees and then combine the results. To test the differences between the HDT and the RF, we have compared the performance of both control and data plane implementations.

The UNSW-NB15 dataset has nine types of attacks, specifically, Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms were used to train our HDT model and evaluate its performance. This dataset comprises 49 network features and a class label. The attack categories Analysis and Backdoor could not be reliably trained and detected by any of the three models used: Decision Trees, Random Forest, and Hierarchical Decision Trees. Due to this unreliability, these attack categories were excluded from testing phases in the evaluations conducted for the data plane and control plane results. The traffic was injected into the BMv2 software switch from a server using Tcpreplay [14] to replay the pcap files.

B. Results

Figure 3 presents the comparison of the control plane performance between HDT, DT, and RF classification models in the context of attack types. The x-axis of the graph shows the categories of attacks and the attack groups, which are the intermediate level in the HDT model. When traffic is classified as an Attack, the next classification level is grouping similar attack types, narrowing the number of classes for the final classification. The criterion for grouping the attack types in this scenario is feature correlation, as attacks that can be classified by common features are grouped together. The y-axis shows the f1-score achieved for every attack type using the different ML models. The results show that HDT outperforms DT and RF in the control plane implementations. Figure 4 demonstrates that the trend observed in the control

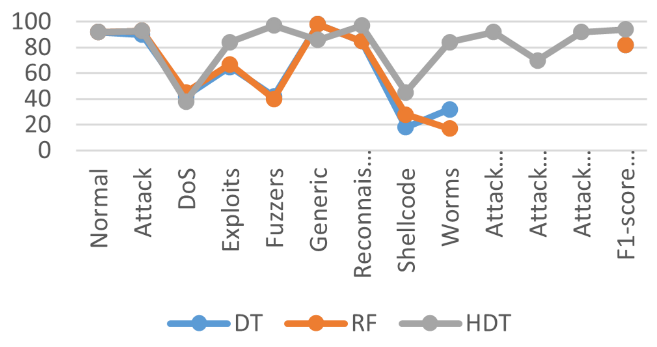


Fig. 3. Comparison of F1 score between Decision Tree (DT), Random Forest (RF), and Hierarchical Decision Tree in the control plane.

plane is consistent in the data plane, where HDT outperforms both the DT and RF baselines. The results indicate that HDT provides higher precision and reliable attack detection when implemented in the data plane, thus making it a promising candidate for real-time threat detection. As highlighted in Table II, HDT implementation yields performance gains from 23% to 26% compared to the flat implementation of a Decision Tree and 11% to 12% compared to the flat RF implementation in the data plane. In the control plane, the precision gain for

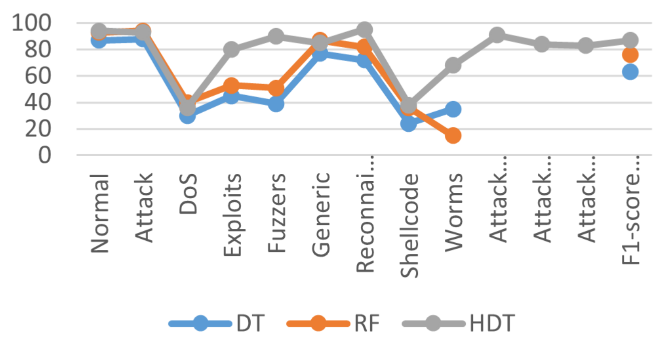


Fig. 4. Comparison of F1-score between Decision Tree (DT), Random Forest (RF), and Hierarchical Decision Tree in the data plane.

TABLE II
COMPARISON OF PERFORMANCE METRICS BETWEEN DATA PLANE AND CONTROL PLANE

	Data Plane			Control Plane		
	DT	**RF**	**HDT**	**DT**	**RF**	**HDT**
Precision Weighted	65	76	88	82	84	94
Recall Weighted	60	75	86	82	81	94
F1-Score Weighted	63	76	87	82	82	94

HDT compared to the flat DT and RF implementations is 12% and 10% respectively.

V. CONCLUSION

We present a Hierarchical Decision Tree (HDT) model and its implementation as an effective approach to enhance ML inference performance in programmable network devices. The experimental results demonstrate better performance in flow-level multiclass classification for HDT when compared to implementations of flat DT and RF models. The design of HDT is highly flexible as it allows different number of nodes and different types and number of features at every level of the HDT. At the same time, it allocates fewer resources to perform classification tasks compared to similar frameworks in the literature. For example, it uses solely the ingress pipeline while other frameworks use both pipelines to take the final classification decision. At the same time, the hierarchical format requires less memory to express the HDT model. Our results show that HDT achieves superior classification accuracy and resource efficiency. Specifically, HDT outperforms traditional flat Decision Tree and Random Forest models. Furthermore, the hierarchical design significantly reduces computational and memory overhead, enabling real-time traffic classification with minimal latency. This makes HDT a promising solution for deployment in high-speed network environments where efficient and accurate traffic classification is critical.

REFERENCES

[1] P.Bosshart, D.Daly, G.Gibb, M.Izzard, N.McKeown, J.Rexford, C. Schlesinger, D.Talayco, A.Vahdat, G.Varghese, D.Walker, “P4: Programming protocol-independent packet processors,” SIGCOMM Comput. Commun. Rev., vol. 44, no. 3, p. 87–95, Jul. 2014.
[2] Z.Xiong, N.Zilberman. “Do switches dream of machine learning? Toward in-network classification. ”HotNets’19: InProceedings of the 18th ACM workshop on hot topics in networks, Nov. 2019.
[3] C.Zheng, N.Zilberman. 2021. “Planter: Seeding Trees within Switches. In SIGCOMM ’21 (Princeton, NJ, USA). ACM, NY, USA, 12–14.”, 2021.
[4] R.Cziva, D.Pezaros. “On the Latency Benefits of Edge NFV”. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 105-106, 2017.
[5] P.Bosshart, G.Gibb, H.S. Kim, G.Varghese, N.McKeown, M.Izzard, F.Mujica, M.Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, ser. SIGCOMM ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 99–110.
[6] G.Siracusano, R.Bifulco. “In-network neural networks”, arXiv preprint arXiv:1801.05731, 2018.
[7] D.Sanvito, G.Siracusano, R.Bifulco. “Can the network be the AI accelerator?” In Proceedings of the 2018 Morning Workshop on In-Network Computing, 2018.
[8] G.Siracusano, S.Galea, D.Sanvito, M.Malekzadeh, H.Haddadi, G. Antichi, R. Bifulco. 2020, “Running neural networks on the nic”, arXiv preprint arXiv:2009.02353, 2020.
[9] C.Zheng,Z.Xiong,T.T.Bui, S.Kaupmees, R.Bensoussane, A.Bernabeu, S.Vargaftik, Y.Ben-Itzhak, and N.Zilberman. “IIsy: practical in-network classification”, arXiv preprint arXiv:2205.08243, 2022.
[10] C.B.Grawitz, R.Meier, A.Dietmüller, T.Bühler, and L.Vanbever. 2019, “pForest: In-network inference with random forests”, CoRR abs/1909.05680. arXiv:1909.05680, 2019.
[11] H.Siddique, M.Neves, C.Kuzniar, I.Haque. “Towards network-accelerated ML-based distributed computer vision systems”,In 2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS). IEEE Computer Society, Los Alamitos, CA, USA, 2021.
[12] X.Zhang, L.Cui, F.P.Tso, W.Jia.“pHeavy: Predicting heavy flows in the programmable data plane”, IEEE Transactions on Network and Service Management 18, 4, 4353–4364, 2021.
[13] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, J. Duan. “Mousika: enable general in-network intelligence in programmable switches by knowledge distillation.” In IEEE INFOCOM 2022 - IEEE Conference on Computer Communications. 2022.
[14] Turner and F.Klassen, “Tcpreplay,” 2013.
[15] T.J.Akem, B.Bütün, M.Gucciardo, M.Fiore, “Henna: hierarchical machine learning inference in programmable switches.” NativeNi ’22: PRocessings of 1st Interational Workshop on Native Network Intelligence, Dec. 2022.
[16] A.Sapio, I.Abdelaziz, A.Aldilaijan, M.Canini, P.Kalnis. “In-network computation is a dumb idea whose time has come.” In HotNets’17 (Palo Alto, CA, USA). ACM, NY, USA.2017.
[17] J.H.Lee, K.P.Singh. “SwitchTree: in-network computing and traffic analyses with random forests.” Neural Computing and Applications, 1–12, 2020.
[18] T.J.Akem, M.Gucciardo, M.Fiore et al.,“Flowrest: practical flow-level inference in programmable switches with random forests,” in IEEE International Conference on Computer Communications, 2023.
[19] K.Razavi, S.D.Fard, G.Karlos, et al,“NetNN: Neural intrusion detection system in programmable networks”, in IEEE Symposium on Computers and Communication, Jun. 2024.
[20] C.Zheng, M.Zang, X.Hong, R.Bensoussane, S.Vargaftik, Y.Ben-Itzhak, and N.Zilberman. 2022. “Automating in-network machine learning.” arXiv, 2022.
[21] X. Jin, X. Li, H. Zhang, R. Soul´e, J. Lee, N. Foster, C. Kim, and I. Stoica, “NetCache: Balancing Key-Value Stores with Fast In-Network Caching,” in Proceedings of the 26th Symposium on Operating Systems Principles, pp. 121–136, 2017.
[22] Z.Yu, Y.Zhang, V.Braverman, M.Chowdhury,and X.Jin, “Netlock: Fast, centralized lock management using programmable switches,” in Proceedings of the ACM SIGCOMM, ser. SIGCOMM ’20. New York, NY, USA: ACM, p. 126–138, 2020.
[23] R.Datta, S.Choi, A.Chowdhary, Y.Park.“P4Guard: Designing P4 Based Firewall.”MILCOM-2018 IEEE Military Communications Conference (MILCOM), 1-6. 2018.
[24] F.P.Tso, G.Hamilton, R.Weber, C.S. Perkins, D.P. Pezaros. “Longer is better: exploiting path diversity in data center networks”,in IEEE International Conference on Distributed Computing Systems (ICDCS), Philadelphia, PA, USA, pp. 430-439, 8-11 Jul 2013.