Empowering In-Network Classification in Programmable Switches by Binary Decision Tree and Knowledge Distillation

Guorui Xie, Qing Li, *Member, IEEE*, Guanglin Duan, Jiaye Lin, Yutao Dong, Yong Jiang, *Member, IEEE*, Dan Zhao, and Yuan Yang

***Abstract*— Given the high packet processing efficiency of programmable switches (e.g., P4 switches of Tbps), several works are proposed to offload the decision tree (DT) to P4 switches for in-network classification. Although the DT is suitable for the match-action paradigm in P4 switches, the range match rules used in the DT may not be supported across devices of different P4 standards. Additionally, emerging models including neural networks (NNs) and ensemble models, have shown their superior performance in networking tasks. But their sophisticated operations pose new challenges to the deployment of these models in switches. In this paper, we propose Mousikav2 to address these drawbacks successfully. First, we design a new tree model, i.e., the binary decision tree (BDT). Unlike the DT, our BDT consists of classification rules in the form of bits, which is a good fit for the standard ternary match supported by different hardware/software switches. Second, we introduce a teacher-student knowledge distillation architecture in Mousikav2, which enables the general transfer from other sophisticated models to the BDT. Through this transfer, sophisticated models are indirectly deployed in switches to avoid switch constraints. Finally, a lightweight P4 program is developed to perform classification tasks in switches with the BDT after knowledge distillation. Experiments on three networking tasks and three commodity switches show that Mousikav2 not only improves the classification accuracy by 3.27%, but also reduces the switch stage and memory usage by 2.00× and 28.67%, respectively. Code is available at https://github.com/xgr19/Mousika.**

***Index Terms*— In-network classification, programmable switch, decision tree, knowledge distillation.**

Manuscript received 24 September 2022; revised 17 May 2023; accepted 5 June 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB3105000, in part by the National Natural Science Foundation of China under Grant 61972189, in part by the Shenzhen Key Laboratory of Software Defined Networking under Grant ZDSYS20140509172959989, and in part by the Research Center for Computer Network (Shenzhen) Ministry of Education. *(Corresponding author: Qing Li.)*

Guorui Xie, Guanglin Duan, Yutao Dong, and Yong Jiang are with the Tsinghua Shenzhen International Graduate School, Shenzhen 518055, China, and also with the Peng Cheng Laboratory (PCL), Shenzhen 518066, China (e-mail: xgr19@mails.tsinghua.edu.cn; duangl16@tsinghua.org.cn; dyt20@mails.tsinghua.edu.cn; jiangy@sz.tsinghua.edu.cn).

Qing Li and Dan Zhao are with the Peng Cheng Laboratory (PCL), Shenzhen 518066, China (e-mail: liq@pcl.ac.cn; zhaod01@pcl.ac.cn).

Jiaye Lin is with the Tsinghua Shenzhen International Graduate School, Shenzhen 518055, China (e-mail: lin-jy22@mails.tsinghua.edu.cn).

Yuan Yang is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: yangyuan_thu@mail.tsinghua.edu.cn).

Digital Object Identifier 10.1109/TNET.2023.3287091

I. INTRODUCTION

RECENT years have witnessed an emerging trend of applying machine learning (ML) to many networking classification tasks [1], [2]. For example, in [3], the authors propose a scheme to classify flows into mice or elephants (i.e., flow size prediction) by utilizing ML models such as Gaussian process regression and neural networks (NNs). In [4], the authors propose a convolutional NN to classify packets into the application types (e.g., browsing, file transferring) that generated them. Thus, a customized routing can be provided to guarantee the differentiated QoS requirements. In [5], ML is also introduced into the malware detection and the authors present a neural framework, FFDNN, to analyze the input traffic and detect malware attacks.

Traditionally, these ML models are implemented in x86 servers to support their complicated ML operations (e.g., floating-point multiplications). Especially, the computation-intensive NNs even require specific GPU-equipped servers for acceleration. As such, traffic has to be redirected to specialized servers for further processing. Although these ML solutions provide promising accuracy, they can cause unsatisfactory throughput and capacity for large-scale data centers and cloud networks [6].

Compared with x86 servers, the modern programmable switches (e.g., P4 switches [7]) support up to Tbps of throughput and enable the programmable logic, which offers the opportunity of deploying in-network ML models. Nonetheless, such a high throughput is achieved by sacrificing the programmable flexibility. Only simple instructions like integer addition/subtraction, and bit shift are allowed in switch actions. These actions also should be triggered by the matched rules in predefined P4 tables, i.e., the match-action paradigm. Besides, each switch has compact resources (e.g., memory and the number of stages) for programmable processing.

To tackle these switch constraints, some works [8], [9], [10], [11], [12], [13] are proposed to deploy the simple rule-based ML model, e.g., the decision tree (DT), in P4 switches for high-speed in-network classification (aka in-network intelligence). For instance, IIsy [8] converts the DT into multiple feature tables and one decision table, i.e., the feature-decision manner. pForest [11] presents the level-table manner, i.e., mapping each level of the DT into a P4 table. Despite the considerable success of these approaches, there are still two key challenges that remain unsolved:

- **The device compatibility**. The DT makes classification decisions by comparing input feature values with thresholds obtained in the training. Previous solutions usually implement this comparison by the range match in P4. However, the range match may not be widely supported, as it is not defined in the core library that must be available in hardware/software programmable switches of different P4 standards [14], [15]. Though IIsy and its extended version [8], [9] present the conversion from the range match to standard match types (e.g., exact/ternary), their authors also state the consequently increased consumption of precious resources in the switch.
- **The model compatibility**. Although the DT seems to fit the switches' match-action paradigm well, its learning capability is not as powerful as other sophisticated ML models (e.g., NNs and ensemble models in [4] and [16]). Due to the aforementioned switch constraints, sophisticated models are difficult to be deployed, resulting in inferior classification accuracy [17]. Despite Planter and its extended version [12], [13] making progress in deploying less sophisticated models like the binary neural network (BNN), the P4 implementation in Planter is manually tuned according to each new added model, which may not scale to diverse models.

Therefore, we propose Mousikav2[1] in this paper, which makes a further step towards enhancing the performance of in-network classification. Mousikav2 supports the indirect deployment of sophisticated models by distilling their classification knowledge into a new proposed rule-based classifier, improving the accuracy of in-network classification while avoiding switch constraints. Unlike the DT, our proposed classifier is natively implemented by the standard ternary match and thus is compatible with devices of different P4 standards without any resource-consuming conversion (e.g., range to exact/ternary in [9]). In summation, **we address the aforementioned challenges with the following key ideas**:

- We redesign the DT to a new rule-based model, the *binary decision tree* (BDT), whose classification rules are bits and thus can be directly encoded by the ternary match in the standard core.p4 library [19]. With the BDT as the deployed model, Mousikav2 is natively supported by most P4 devices.
- We adopt a teacher-student *knowledge distillation* architecture to train the BDT. That is, the classification knowledge of diverse sophisticated teacher models (e.g., NNs or ensemble models) is distilled and transferred into the BDT to avoid deployment constraints and improve classification accuracy.
- We design a delicate *P4 program* to use the classification rules of the BDT. This program only takes up two tables and two stages of the switch, which is lightweight enough for compact switch resources.

[1]This work was presented in part at the 2022 IEEE International Conference on Computer Communications (INFOCOM) [18].

We conduct thorough experiments on three classification scenarios (flow size prediction, traffic type classification, and malware detection) and three commodity P4 switches to evaluate the performance of Mousikav2. The experimental results reveal that: **1)** The BDT after knowledge distillation usually has a better classification performance. For the task of traffic type classification, the BDT after knowledge distillation improves the accuracy of the DT by 3.27% (97.66% vs. 94.39%) **2)** Knowledge distillation is helpful to reduce the training time and classification rules of the BDT. E.g., the training time and the number of classification rules are reduced by 6.01$\times$ and 4.28$\times$, respectively. **3)** Due to the efficient processing performance of the hardware switch, deploying Mousikav2 in the three switches has little impact on their packet forwarding. For the traffic speed of 100Gbps, the traffic can still be transmitted at line rate (the packet latency is $\sim$ 660 nanoseconds). **4)** Compared with the existing DT implementation, Mousikav2 only occupies a small amount of switch resources. For the malware detection task, Mousikav2 only takes up 2 stages and 5.20% of the TCAM, which is 2.00$\times \downarrow$ and 28.67% $\downarrow$ compared with IIsy [8].

II. Background

A. Machine Learning for Networking Classification

In recent years, machine learning has been employed in every possible field to leverage its amazing learning power, e.g., computer vision and natural language process [20], [21], [22], [23]. The networking field has also seen several schemes proposed to exploit ML for networking classification tasks [1].

In [3], the authors concern with the problem of predicting the size of a flow and detecting elephant flows (very large flows). They describe the problem as a learning-based classification task and employ machine learning models like Gaussian process regression (GPR) and neural network (NN) for flow size prediction. In [4], the authors focus on the problem of classifying Internet traffic by application type. They designed an NN-based system that can classify IP packets into application protocols (e.g., FTP and P2P) or applications (e.g., web browsing and file transferring). The works in [16] and [24] also focus on employing the power of ML for the traffic classification, the employed models are recurrent NN and one-dimensional convolutional NN, respectively. In [5], the authors propose a feed-forward deep NN (FFDNN) and utilize 48 statistical features of flows (e.g., the average packet sending rate and the inter-packet arrival time) for detecting malware attacks. In [17], the authors utilize different numbers of the fully-connected neural layers to build two models, i.e., ANN and DNN. Both models show superiority in the task of malware detection.

Now, many ML-based solutions have reached promising accuracy on networking classification tasks [2]. Based on them, network administrators can yield many management and security gains, e.g., offering differentiated QoS provision by the traffic type classification and defending against attacks by the malware detection. Nevertheless, the current solutions of ML deployment require high-performance x86 servers to support the complicated ML operations. Not to

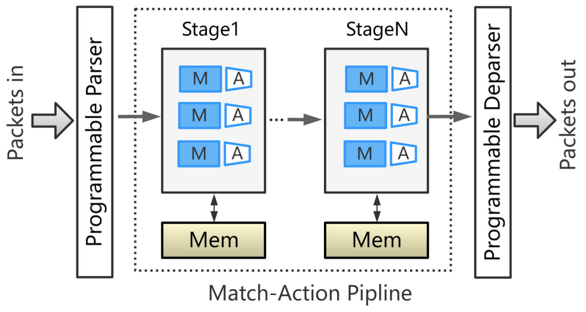


Fig. 1. The general architecture of a P4 switch.

mention that computation-intensive NNs even rely on specific GPUs for acceleration. Hence, traffic must be redirected to specialized servers for analysis, which induces significant processing latency and severely degrades throughput in large-scale commercial networks [6].

B. P4 Switch and Its Constraints

Recently, P4 switches of Tbps have been deployed in several commercial data centers [25], [26]. Besides the high throughput, P4 switches also support the programmable logic, which capacitates the direct deployment of ML models inside network devices.

There exist various software and hardware switches that follow different P4 standards [14], [15]. Fig. 1 demonstrates the general architecture of a switch's data plane, i.e., the protocol independent switch architecture (PISA). As shown, the arriving packet is first mapped into a packet header vector (PHV) by the programmable parser. Then, the PHV is passed to a match-action pipeline. The pipeline consists of match-action units (MAUs) arranged in N stages (typically, $N = 12$). Each stage is assigned a memory block (mem). If a header field (e.g., the destination port) in the PHV matches (M) a given table entry (stored in mem), further processes in the action unit (A) associated with the matched entry are triggered. Finally, the processed PHV is reorganized into a packet by the programmable deparser. In summary, PISA allows network administrators to define customized processes (e.g., match-action tables) in P4 language and then instantiate them inside MAUs. This novel architecture empowers P4 switches with a packet processing throughput of Tbps. However, a shortcoming of the architecture is that it has rather limited programmable capability. Now we discuss three main programmable constraints below.

Processing constraints. To guarantee the high-speed processing, complex instructions such as multiplications, divisions, and other floating-point operations (e.g., polynomials or logarithms) are not supported [8], [27]. Packet processing in MAUs is limited to very simple instructions like integer additions and bit shifts. Besides, the number of supported instructions in a specific action is also limited. As a result, it is difficult to deploy most ML models that require complicated floating-point operations in P4 switches.

Matching constraints. For P4 language, the standard library (i.e., core.p4 [19]) defines three kinds of matching which are widely supported by diverse devices: **1)** Exact match. The input key (e.g., headers in the PHV) has to match exactly with the field in the rule. **2)** Ternary match (wildcard). The input key is AND with the Mask associated with each rule, and then compared with a corresponding Value for an exact match. **3)** Longest prefix match (LPM). Compared with the ternary match, this case guarantees that the Mask is a series of consecutive bits 1 followed by a series of consecutive bits 0 [28]. Other libraries (e.g., v1model.p4 [14]) may define additional match kinds such as range match and fuzzy match. But these additional match kinds may not be available in many switch targets [8].

Resource constraints. Each stage is evenly equipped with two high-speed types of memory. One is TCAM, which is a content-addressable memory suitable for fast table lookups. TCAM is used to store table entries with match kinds including ternary, LPM, and range match [29]. The other is SRAM which is used to store exact match table entries and stateful registers. Unfortunately, the total amount of memory in the switch is small. The amount of SRAM is in the order of 100MB, while TCAM is far less than that [30]. Furthermore, the number of available switch stages is also small (typically 12 [8]) as passing too many stages will delay packet forwarding. As such, many networking tasks (e.g., NAT, fault tolerance, and load balancing) have to compete to share these precious resources.

C. In-Network Classification

Given the discussed switch constraints, an emerging trend is to implement the rule-based ML model, e.g., decision tree (DT) [31], in switches for high-speed processing, i.e., in-network classification (aka in-network intelligence) [8], [9], [10], [11], [12], [13].

There are two main ways to map tree models into P4, i.e., the *level-table* manner and the *feature-decision* manner. In SwitchTree [10] and pForest [11], the authors use the level-table manner to map each level of the DT into a match-action table. A DT node in one level table compares a specific feature against the threshold, assigning the node ID of the next level table to be matched. As level tables from different DTs are independent, they can be placed in the same switch stage to be executed in parallel, further forming the ensemble classifier (i.e., random forest, RF) for accuracy gains. However, the level tables from the same DT should be executed sequentially in different stages, causing the number of consumed switch stages to be proportional to the depth of the DT [9].

IIsy and its extended version [8], [9] overcome this stage-depth dependence by the feature-decision manner. That is, each feature used in a DT is encoded to be a match-action table, being responsible to compare all thresholds of this feature and outputting the compared results. Finally, a decision table is utilized to output classification results according to the compared results. In an ideal case (where the DT is small), IIsy consumes only two stages as feature tables can be conducted parallelly in one stage. Nevertheless, as the DT grows larger, these feature tables may overflow several stages [11]. Besides, as the range match used for threshold comparisons is not compatible well with devices from different P4 standards, IIsy proposes to convert the feature comparison into standard

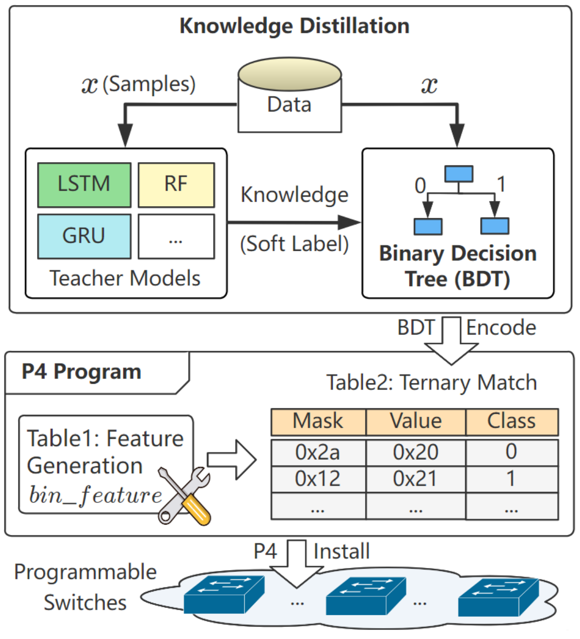


Fig. 2. The Mousikav2 framework.

match types (exact/ternary/LPM), resulting in extra resource consumption of compact switches [8].

Planer and its extended version [12], [13] aim to automate and modularize the in-network classification, maintaining modules of the encoding-based (feature-decision) and the direct-mapping (level-table) for the DT deployment. Also, modularization makes it possible to extend the supported ML models. As an illustration, Planter shows the way to support models such as BNNs which mainly depend on the bit operations, through newly added modules of XNOR and PopCount. However, precisely designing modules for diverse models is challenging for human resources. Additionally, due to the inherent switch constraints, it remains difficult to support floating-point NNs with sophisticated operations.

Hence, we propose Mousikav2, which boosts the in-network classification/intelligence in two aspects. First, as the goal of extending ML models (e.g., NNs and RF) is the accuracy gain, we propose an indirect deployment of these models, i.e., transferring their knowledge to a simple model for a unified deployment. Second, we design the BDT as the targeted simple model that natively uses the standard ternary match for resource-efficient deployment in different P4 devices, without the costly match conversion.

III. Mousikav2 Overview

Fig. 2 presents the Mousikav2 framework. It mainly contains three key components: the binary decision tree (BDT), the teacher-student knowledge distillation architecture to train the BDT, and the P4 program to install the trained BDT.

The BDT (detailed in Section IV) has many inner and leaf nodes. Each inner node checks one bit of the sample features and routes the sample to the left subnode (if the bit value is 0) or the right subnode (if the bit value is 1). When the sample is routed to a leaf node, the predicted class label is obtained. **The knowledge distillation (detailed in Section V)** leverages sophisticated ML models to train (teach) a BDT. Many powerful models can be employed as the teacher model, including but not limited to different NNs (e.g., LSTM [32] and GRU [33]) and ensemble models (e.g., RF [34]). The teacher model is trained in advance. Then, for each sample x in the dataset, the teacher model transfers its learned knowledge (aka soft label) to the BDT, supervising the growth of inner and leaf nodes. **The P4 program (detailed in Section VI)** mainly consists of two match-action tables (two separate switch stages between the programmable parser and deparser). Table1 has no table entries and its default action is to map the features (e.g., ports, packet size in the PHV) of a packet into the bit string $bin\_feature$. Then, Table2 classifies $bin\_feature$ according to the ternary match entries (i.e., encoded rules of the BDT). Each entry contains three fields, i.e., Mask, Value, and Class. As discussed in Section II-B, the ternary match is performed by finding the matched Value after $bin\_feature$ AND Mask and returning the corresponding Class label.

Mapping packet features into bits is partially similar to the feature binning [35]. Feature binning maps feature values into different buckets to introduce non-linearity for accuracy gains. However, the range match is required to decide which bucket a specific feature value belongs to. Also, [35] needs the extra training of the DT to decide the range per bucket for the subsequent model classification.

IV. Binary Decision Tree

In this section, we first introduce the BDT training algorithm which is the base of our knowledge distillation. Then, we use an example to demonstrate the training algorithm intuitively.

A. BDT Training

Based on DT [31], we design Algorithm 1 that grows a BDT on the given networking classification dataset D. Training samples (packets) in D have the form of (x_i, y_i). x_i is a bit string of length $|A|$, i.e., $bin\_feature$ of Section III. $y_i \in \mathbb{R}^K$ denotes the one-hot representation of the class label (K-dimensional vector,[2] where the value of the corresponding dimension of the category is 1, and the remaining are 0). The main parts of the BDT training algorithm are:

- Lines 3 $\sim$ 6 indicate that if all samples in D belong to the same category ($\text{Cls}(y) \equiv C$), the BDT generation process stops and the current node N is regarded as a leaf with class label C. Here, function Cls(.) finds the corresponding class label in y:
$$\text{Cls}(y) = \arg\max_j \{y^1, \ldots, y^j, \ldots, y^K\}, \qquad (1)$$
where $j \in [1, K]$ is the index that has the value of 1 in the one-hot vector y.
- Lines 7 $\sim$ 10 reflect another stop processing: if the current bit set A is empty (the bits in A are removed one by one during the generation process, see Line 19) or all samples have the same value (0 or 1) on each bit in A, the current node N is also denoted as a leaf and its class label $N.cls$ is assigned by the returned value of CntMaxCls(.).

[2]Using vectors to represent class labels is convenient for the following knowledge distillation.

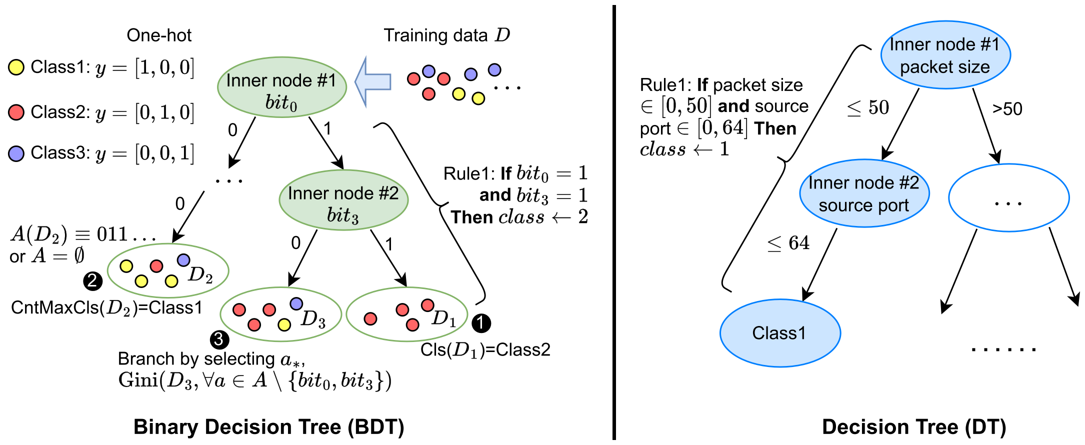

One-hot
Class1: $y=[1,0,0]$
Class2: $y=[0,1,0]$
Class3: $y=[0,0,1]$
Training data D
Inner node #1 bit_0
Inner node #2 bit_3
$A(D_2)\equiv 011\ldots$ or $A=\emptyset$
CntMaxCls(D_2)=Class1
Rule1: If $bit_0=1$ and $bit_3=1$ Then $class\leftarrow 2$
Cls(D_1)=Class2
Branch by selecting a_*, Gini($D_3,\forall a\in A\setminus\{bit_0,bit_3\}$)
Binary Decision Tree (BDT)
Rule1: If packet size $\in[0,50]$ and source port $\in[0,64]$ Then $class\leftarrow 1$
Inner node #1 packet size
Inner node #2 source port
≤ 50
>50
≤ 64
Class1
Decision Tree (DT)

Fig. 3. The binary decision tree (BDT) and the ordinary decision tree (DT).

Algorithm 1 BDT Training Without Knowledge Distillation

```
Input: Training set $D = \{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$,
    Bit features $A=\{a_1,a_2,\ldots,a_m\}$.
1: function BDTGENERATE($D$, $A$)
2:     Generate node $N$;
3:     if $\forall y\in D, \mathrm{Cls}(y)\equiv C$ then
4:         $N.cls \leftarrow C$;                          // Leaf node
5:         return;
6:     end if
7:     if $A=\emptyset$ or $\forall x_1,x_2\in D, x_1.A\equiv x_2.A$ then
8:         $N.cls \leftarrow \mathrm{CntMaxCls}(\forall y\in D)$;      // Leaf node
9:         return;
10:    end if
11:    Select the optimal bit $a_*$ by $\mathrm{Gini}(D,\forall a\in A)$;
12:    for $v\in\{0,1\}$ do
13:        Create a branch of node $N$ as $N.brc$;
14:        $D_v \leftarrow \{(x,y)\mid x.a_*=v,\ x\in D\}$;
15:        if $|D_v| \leq min\_samples\_leaf$ then
16:            $N.brc.cls \leftarrow \mathrm{CntMaxCls}(\forall y\in D_v)$;   // Leaf
17:            return;
18:        else                                       // Inner node
19:            $N.brc \leftarrow$ BDTGENERATE($D_v, A\setminus\{a_*\}$);
20:        end if
21:    end for
22: end function
Output: The trained binary decision tree.
```

CntMaxCls(.) is a function to generate a class label from all $y\in D$:

$$P=(p_1,\ldots,p_j,\ldots,p_K) = \frac{1}{|D|}\sum_{y\in D} y, \tag{2}$$

$$\mathrm{CntMaxCls}(\forall y\in D)=\arg\max_j P, \tag{3}$$

where P is the vector maintaining the probability of each class, and the index j of the class with the maximum probability is returned. Note that y is one-hot [36], i.e., $\sum^K y_i = 1.0$.

- Line 11 selects the optimal branch bit a_* by the Gini index criterion [31]:

$$a_* = \arg\min_{a\in A} \mathrm{Gini}(D,a), \tag{4}$$

$$\mathrm{Gini}(D,a)=\sum_{v\in\{0,1\}}\frac{|D_v|}{|D|}(1-P_{D_v}^\top P_{D_v}), \tag{5}$$

where $D_v=\{(x,y)\mid x.a=v,\ x\in D\}$ contains all samples whose value of bit a is v, P_{D_v} is calculated by Equation 2 denoting the class probability vector of subset D_v, and the optimal branch bit a_* is the one that has the minimum Gini index. The minimum Gini index reflects the desired effect in the splitting, i.e., the samples from the same class are likely to be assigned in the same node.
- Lines 12 $\sim$ 21 are the loop for tree branching. For each value $v\in\{0,1\}$ of the optimal a_*, if the corresponding subset D_v contains samples fewer than a predefined threshold (e.g., $min\_samples\_leaf = 5$ in our experiments), the tree generation stops (Lines 15 $\sim$ 17). Otherwise, the tree will grow on the new subset D_v and new bit set $A=A\setminus\{a_*\}$ (Line 19).

B. Training Example

The left side of Fig. 3 depicts how Algorithm 1 grows a BDT. As shown, there are samples of three classes (illustrated as red, purple, and yellow circles, respectively) in the training dataset D, where the label y of samples in a class is represented by a one-hot vector. Then, all training samples are recursively split into nodes until leaf nodes are reached. We summarize three cases (❶ $\sim$ ❸) during this splitting:

- In ❶, the samples in D_1 are from the same class (i.e., the Class2). In light of Lines 3 $\sim$ 6 in Algorithm 1, we set the node of D_1 as a leaf node, stopping the splitting of D_1.
- In ❷, D_2 holds samples of different classes. As there have been several times of sample splitting (node branching) before D_2, the current bit set A may be empty or each sample has the same bit values (i.e., $A(D_2)\equiv 011\ldots$). Thus, we also set the node of D_2 as a leaf node by Lines 7 $\sim$ 10 in Algorithm 1 with Equation 2 and 3. Apparently, CntMaxCls(D_2) returns Class1 as Class1 has the maximum probability.

- In ❸, we continue to split the samples in current inner node by Lines 11 $\sim$ 21 in Algorithm 1. Notably, as inner nodes #1 and #2 have used bit_0 and bit_3, these two used bits are removed in the current splitting.

After the BDT is trained, we can extract several rules of bits. For example, the Rule1 in the left side of Fig. 3 utilizes bit_0 and bit_3 to classify the new samples. As discussed, the bits are from the selected packet features (e.g., ports, packet size). The right side of Fig. 3 depicts a trained ordinary DT. During the training, the DT also uses the Gini index to choose optimal features and their thresholds for sample splitting [31]. Compared with the DT, the input of the BDT is bits, which makes the Gini index calculation much faster in the training. E.g., in the right side of Fig. 3, if the packet size has m bits, the complexity of the Gini index calculation in the DT is $O(2^m)$ as all candidate thresholds must be compared in the ordinary DT to find the optimal (i.e., 50 in the right side of Fig. 3). But the BDT only needs to calculate $O(m)$ times as only bits are considered. Also, according to Line 12 in Algorithm 1, the BDT uses bits of 0 or 1 rather than ranges as branch conditions to split samples. Hence, from the root to each leaf, there is a classification rule consisting of zeros and ones, which is a good fit for the well-supported ternary match in different switches (see Section VI-A for encoding the BDT into ternary match entries).

V. Knowledge Distillation

In this section, we first introduce the preliminaries of knowledge distillation, and then adapt knowledge distillation for our BDT.

A. Preliminaries

Due to the limited computation capacity and memory of mobile devices, deploying sophisticated ML models in these devices encounters great challenges. To this end, the idea of learning a lightweight student model from a sophisticated teacher model is formally popularized as knowledge distillation [37]. In [38], the authors define the class probability vectors of samples predicted by the teacher model as the "knowledge" (i.e., the soft label). Then, the teacher model directly supervises the gradient descent training of the student model on a transfer dataset through the following soft loss function L_{soft}:

$$L_{soft} = -\sum s_i \log(p_i), \tag{6}$$

where for a training sample x_i, the soft label of the teacher is s_i, and the student predicted probability vector is p_i. However, superior teacher models also have a certain error rate by using s_i to predict the class labels of samples. Hence, the ground truth (aka the hard label, h_i), i.e., the one-hot vector in Section IV (representing the true class label) is also considered in the student training:

$$L_{hard} = -\sum h_i \log(p_i). \tag{7}$$

Then, the final gradient descent loss L is the weighted sum of L_{soft} and L_{hard}, that is:

$$L = \alpha L_{soft} + \beta L_{hard}, \tag{8}$$

where weights α and β are in the range of (0, 1), $\alpha + \beta = 1$.

The great success in practice shows that the student model can mimic the classification behaviors of the teacher model and obtain a comparable or even superior performance [39], [40], [41], [42]. For example, the authors in [42] present a tree-structure neural network, i.e., the soft decision tree (SDT), as the student model. After the knowledge distillation-based gradient descent, SDT yields a competitive performance on image classification. Furthermore, the interpretability of the SDT can be partly shown by tracing the classification paths in the tree.

B. Knowledge Distillation in BDT

The conventional knowledge distillation requires the student model to be parametric and optimizable by the gradient descent. However, our BDT is a rule-based classifier with no parameters to be optimized. Fortunately, some attempts at knowledge distillation without gradient descent seem promising. For instance, the authors in [43] propose the rectified decision tree (ReDT). In ReDT, the soft labels of the teacher model incorporate the impurity calculation to determine the feature selection and node splitting. Following these efforts, we can adapt the conventional knowledge distillation for our rule-based BDT without gradient descent.

For a K-class classification problem, we still use the class probabilities output of the teacher model as the soft label (i.e., the knowledge). The sample x_i in the transfer dataset is first fed to the trained teacher model to output the soft label $s_i \in \mathbb{R}^K$. Besides, each sample has a hard label h_i, i.e., the one-hot vector indicating its true class label (ground truth). Then, we denote

$$\hat{y_i} = \alpha s_i + \beta h_i, \tag{9}$$

where $\hat{y_i}$ is the weighted sum of the soft and hard labels. The pairs of $\{(x_1, \hat{y_1}), \ldots, \{(x_i, \hat{y_i}), \ldots, (x_n, \hat{y_n})\}$ are formed the training set D in Algorithm 1 to train a BDT by knowledge distillation. That is, all we need to do for training the BDT by knowledge distillation is to change the algorithm input (replacing the one-hot vector y_i by the weighted label $\hat{y_i}$) while leaving everything else in the algorithm intact.

Though the alteration is straightforward, the learned knowledge of the teacher model deeply influences the growth of the BDT. Formally, the class probability distribution in a dataset is changed from Equation 2 to

$$\hat{P} = (\hat{p_1}, \ldots, \hat{p_j}, \ldots, \hat{p_K}) = \frac{1}{|D|} \sum_{\hat{y} \in D} \hat{y}, \tag{10}$$

where $\hat{y}$ is the weighted sum of class probabilities with soft and hard labels in Equation 9. Accordingly, the calculation of Equation 3 $\sim$ 5 is also changed. In other words, by introducing $\hat{y}$, the knowledge from the teacher can cooperate with the ground truth, teaching the BDT where to stop branching (Lines 7 $\sim$ 10 in Algorithm 1 with changed Equation 2 and 3) and which is the optimal branch bit (Line 11 with changed Equation 4 and 5). Training by knowledge distillation helps the BDT to mimic the classification of the teacher model and obtain a relatively high performance [42], [43]. In addition, Mousikav2 supports a wide variety of models as the teacher, ranging from different NNs to ensemble models, provided that they have good performance and can output class probabilities as the soft labels.

VI. P4 Program

In this section, we discuss the P4 program for in-network classification. First, we introduce how to encode the BDT rules into the ternary match table entries. Then, we install two P4 tables in a switch, utilizing the encoded entries for classification tasks.

A. BDT Encoding

According to the left side of Fig. 3, the classification rules of a trained BDT consist of zeros and ones. Starting from the first inner node (i.e., the root), a bit feature is checked to be zero or one and the corresponding left or right branch is selected. This branching procedure is repeated until a leaf node is reached, which maintains the predicted class label. In other words, we only need to check the bit value in specific positions according to the traversed inner nodes, which is similar to the ternary match and makes it easy to transfer a BDT classification rule to a ternary match table entry. For example, let $x = bit_0 bit_1 \ldots bit_7$ denotes one input sample which has 8 bits. A rule output by the BDT in Fig. 3 is:

$$\textbf{If } bit_0 = 1 \textbf{ and } bit_3 = 1 \textbf{ Then } class \leftarrow \text{Class2}.$$

As bit values in positions 0 and 3 need to be checked, the ternary Mask is 0b10010000 and the corresponding ternary Value is 0b10010000. We just need to examine whether x AND Mask equals the ternary Value to validate the rule matching. If it does match (i.e., x AND Mask $=$ Value), the class label (Class2) will be used as a parameter in the assignment action. Therefore, we fit the sequential decision process of a BDT into the ternary match and action of switch tables. Notably, there is no accuracy decrement after converting the BDT into the ternary match entries (see Section VII-D).

B. Switch Tables

As shown in Fig. 2, after encoding the BDT classification rules as ternary match table entries, we develop a P4 program of two switch tables to perform networking classification.

As shown in Listing 1, the first table *tb_concat_feature* is for *bin_feature* initialization. *tb_concat_feature* uses default action *ac_concat_feature* to initialize *bin_feature* that is stored in the PHV's metadata. As the parsed packet is also stored in the PHV (in the form of bits), a packet header can be easily assigned to the corresponding position of *bin_feature*. For instance, Line 3 in Listing 1 assigns the IP protocol field of the parsed IPv4 packet to *meta.bin_feature*'s low 8 bits.

Listing 1. P4 code fragment that initializes *bin_feature*.

```
1  // assign specific field to bin_feature
2  action ac_concat_feature() {
3  meta.bin_feature[7:0] = hdr.ipv4.protocol;
4  // the assignment of other packet headers is omitted \ldots
5  }
6  // stage~1: feature initialize
7  table tb_concat_feature{
8  actions = {
9  ac_parse_bin_feature;
10 }
11 default_action = ac_parse_bin_feature;
12 }
```

TABLE I
An Example of Matched Entry in *tb_packet_cls*

Mask	Value	Class (Port)
0x0001 0022 0250 0000 0000	0x0000 0022 0210 0000 0000	1

To speed up the operations, P4 switches allow separate tables allocated in the same stage have simultaneity. But we find that tables of *bin_feature* initialization and classification can not be performed in parallel at the same stage, as their operations have a dependence on *bin_feature*. That is, *bin_feature* should be first initialized and then used as the key for classification. To resolve such a data dependency, we place the second table of a specific classification task at an additional switch stage (thus a total of 2 stages in our P4 program) and introduce it as follows.

With the table entries encoded from a BDT, the classification process is transformed into the ternary match-action of table *tb_packet_cls* in Listing 2. According to Line 8 in Listing 2, if there is a match, the corresponding action *ac_packet_forward* (Lines 2 $\sim$ 4 in Listing 2) will be triggered. In *ac_packet_forward*, we map classes to forwarding ports of the switch, and packets of different classes will be forwarded to different ports by the switch. For instance, for a coming packet x, if its $bin\_feature = 0x0000\ 0022\ 0210\ 0000\ 0000$, it will match (i.e., $x.bin\_feature$ AND Mask $=$ Value) the entry shown in Table I. Then the $port = 1$ of the matched entry is passed to *ac_packet_forward* to label the forwarding port (*ucast_egress_port*) of this packet.

Listing 2. P4 code fragment that implements classification.

```
1  // forward packets to different ports
2  action ac_packet_forward(PortId_t port){
3  ig_tm_md.ucast_egress_port = port;
4  }
5  // stage~2: BDT-based classification
6  table tb_packet_cls {
7  key = {
8  meta.bin_feature: ternary;
9  }
10 actions = {
11 ac_packet_forward;
12 }
13 size = 1024;
14 }
```

Actually, we only provide an example of operations for the classified packets in $ac\_packet\_forward$, and other operations can be easily adjusted according to the specific scenario. E.g., the network administrator can change the match entries so that packets of different predicted classes are forwarded to ports with predefined quality-of-service provisions.

VII. Evaluation

In this section, we first introduce the experimental settings. Then, the classification performance of the BDT, DT, and knowledge distillation is evaluated. Finally, we analyze the efficiency of Mousikav2 with different commodity switches and traffic speeds.

A. Experimental Settings

Networking tasks and datasets. To demonstrate the performance of Mousikav2, we build it to classify specific target classes within the context of three tasks:

- We utilize the UNIV1 dataset made available in [44] for the flow size prediction. UNIV1 dataset is collected in one university campus data center. During the classification in this work, we classify the packets that belong to the top 20% flows (w.r.t flow size) in UNIV1 as elephants, while the other packets are mice.
- We utilize the ISCX dataset [45] for the task of classifying traffic according to the application types (i.e., traffic type classification). ISCX dataset contains packets of six application types (Email, Chat, Streaming, File Transfer, VoIP, P2P).
- We leverage the Bot-IoT dataset [46] for the malware detection, identifying whether the packets are from legitimate activities or malicious attacks (e.g., DDoS and service scanning).

In all datasets, we split their traffic into five subsets for the 5-fold cross validation. That is, experiments are run five times, and only one particular subset is used for testing per time, while other subsets are used for training.

Hardware and simulated traffic. Model training, knowledge distillation, and tree model encoding are conducted on a high-performance server with Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and NVIDIA GPU RTX 2080 Ti. Besides, Mousikav2 is deployed in three commodity P4 switches, i.e., EdgeCore Wedge 100BF-65X,[3] H3C S9850-32H,[4] and OpenMesh BF-48 $\times$ 6Z[5] for traffic testing. The tested traffic of 100Gbps is generated by the traffic generator of KEYSIGHT XGS12-SDL[6] under the Internet mix (IMIX) mode. The IMIX mode generates traffic of hybrid-length packets, which is expected to approximate the real-world traffic [47].

Model settings. Teacher models include two ensemble models (RF [34], GBDT [48]) and three NNs (GRU [33], LSTM [32], MLP [5]). These teacher models are trained by the off-the-shelf ML libraries, i.e., Scikit-learn [50] and Pytorch [51]. As models are of multiple hyperparameters, we also use the grid search to find their optimal settings per dataset. For example, batch size (100 $\sim$ 1000) and learning rate (0.1 $\sim$ 0.00001 of NNs are searched in the training to find the best accuracy. Notably, the current ML libraries do not support training BDT by knowledge distillation. Hence, we implement[7] the BDT and its knowledge distillation from scratch in Python 3.6 with the acceleration of Numpy 1.19.3.[8] Learning from [8], we select a set of packet header fields as the model input, including IP protocol, time to live (TTL), and packet size. As suggested by [16] and [52], some bias fields (e.g., IP/MAC addresses) and meaningless fields (e.g., checksum) are not considered here.

α and β. To perform knowledge distillation, we first need to decide the summation ratios of the soft label and the hard label. Fig. 4 shows performance metrics of the distilled BDT with different α and β. As $\alpha = 0.25, \beta = 0.75$ has the best performance of all three tasks, we use this setting for the following experiments.

B. Classification Performance

Fig. 5 demonstrates the classification accuracy on tasks of flow size prediction, traffic type classification, and malware detection. Among the three tasks, the DT is slightly better than the BDT, because converting the features into the form of bits actually reduces the useful information for classification. However, after the knowledge distillation from superior teacher models, the BDT outperforms the DT. As an illustration, in Fig. 5b, accuracies of the DT and the BDT are respectively 94.39% and 93.29% on the traffic type classification. But the BDT overtakes the DT by 3.27% (97.66% vs. 94.39%) after the knowledge distillation from the RF.

We also consider other three classification metrics in this paper, i.e., the F1-score in Fig. 6, the precision in Fig. 7, and the recall in Fig. 8. Overall, we achieve a similar conclusion. That is, with the sophisticated ML models as the teacher, we can train a better BDT on networking tasks. Besides, we also find some interesting cases in these figures: 1) The BDT distilled from a sophisticated teacher can perform worse. For example, in Fig. 7a, the precision of the BDT after knowledge distillation from the GBDT is lower (0.42%$\downarrow$) than the precision of the original BDT. In [53], [54], the authors also find that students distilled from a bigger teacher may not perform better. 2) The BDT can outperform its teacher. In Fig. 5b, the accuracy of the distilled BDT and its teacher RF is 97.66% and 95.77%, respectively. Works [55], [56] also report that students can outperform more sophisticated teachers. These cases are open problems and two reasonable hypotheses are: When trained with distillation, the teacher's knowledge of some classes is incomprehensible to the student, resulting in a poor student [53];

[3]https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=334

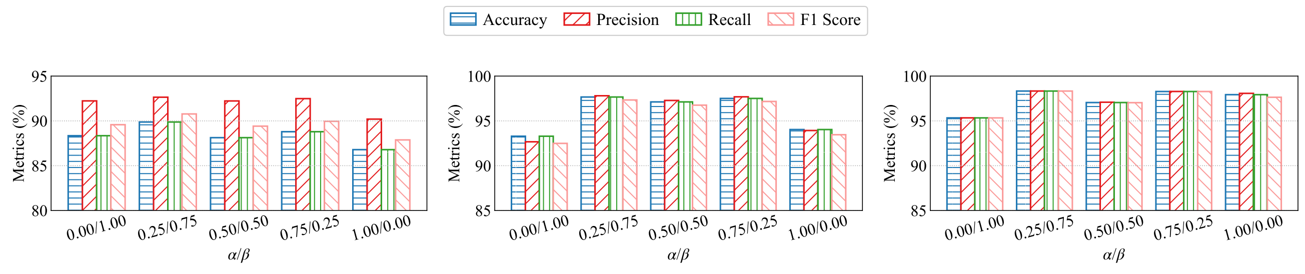
[4]https://www.h3c.com/en/Products_Technology/Enterprise_Products/Switches/Data_Center_Switches/H3C_S9850/

[5]http://www.tooyum.com/products/OpenMesh_BF48 × 6Z.html

[6]https://www.keysight.com/us/en/products/network-test/network-test-hardware/xgs12-chassis-platform.html?rd=1

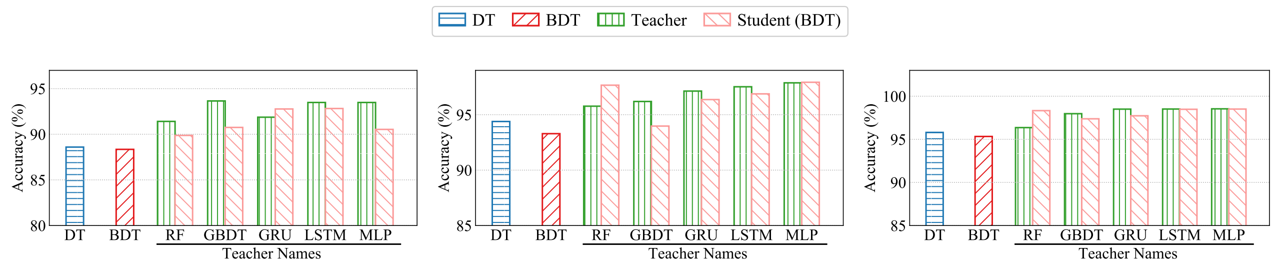
[7]We have optimized our code so that it is much faster than the previous version presented in [18].

[8]https://numpy.org/



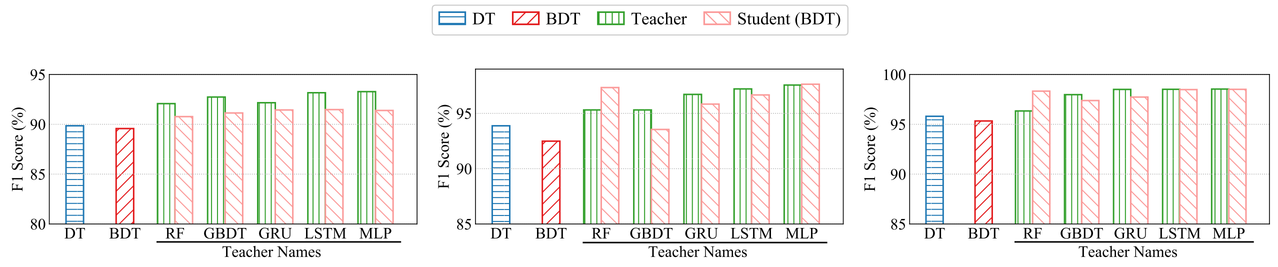
(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 4. The BDT classification performance of different values on α and β (the teacher model is RF).



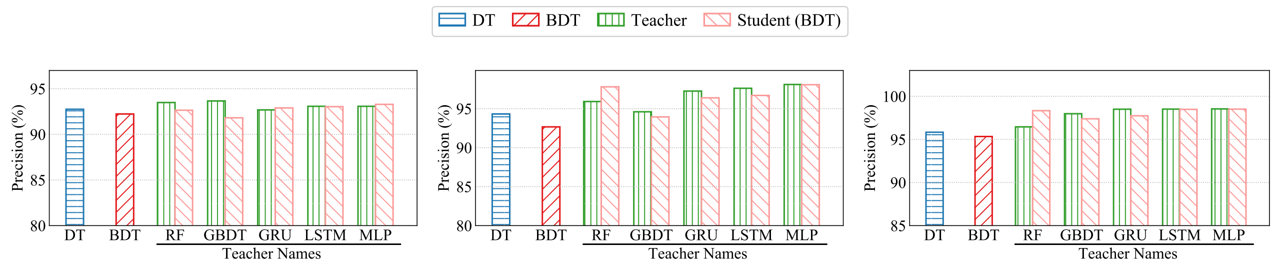
(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 5. The classification accuracy of different models on three networking tasks.



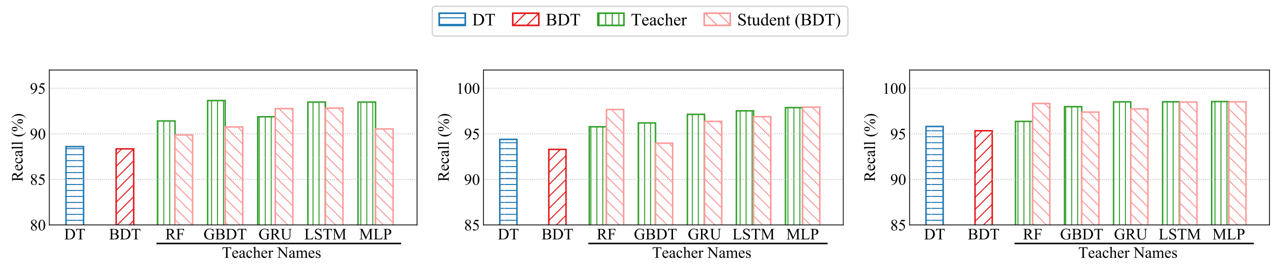
(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 6. The classification F1-score of different models on three networking tasks.



(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 7. The classification precision of different models on three networking tasks.



(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 8. The classification recall of different models on three networking tasks.

Hard labels can correct wrong predictions of the imperfect teacher, which helps to train a stronger student than the teacher [57].

C. Training Time and Classification Rules

Fig. 9 and 10 depict the training time and the number of classification rules in the BDT. As shown, knowledge

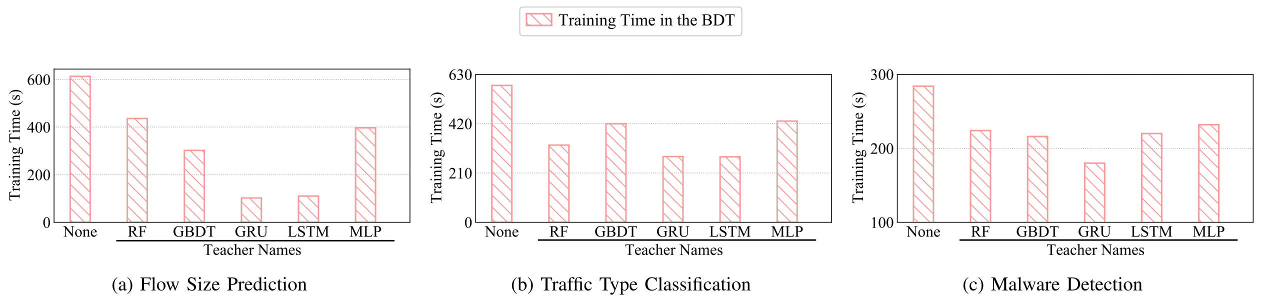


Fig. 9. The training time of the BDT without or with knowledge distillation on three networking tasks.

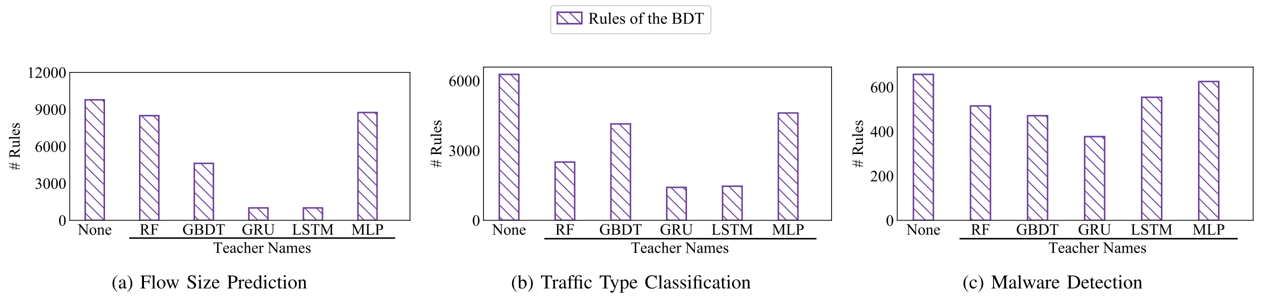


Fig. 10. The number of classification rules in BDTs without or with knowledge distillation.

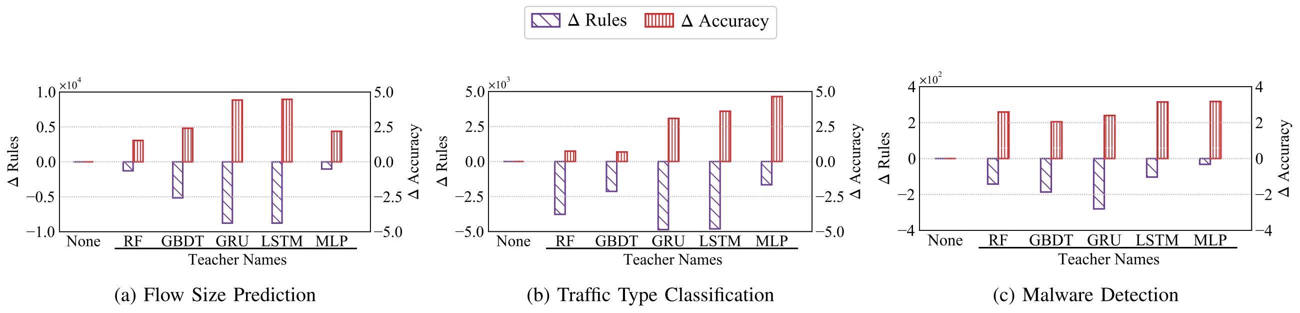


Fig. 11. The difference of rules and accuracies between the distilled BDT and original BDT ($\Delta = distilled - original$).

distillation not only improves the classification performance, but also reduces the training time and the number of classification rules in the BDT. For instance, in Fig. 9a, the training time of the BDT after the knowledge distillation from the GRU is 102 seconds, which is 6.01$\times$ faster than that of the original BDT (613 seconds). In Fig. 10b, the knowledge distillation from the LSTM helps to reduce the number of classification rules by 4.28$\times$ (i.e., from 6273 to 1466). One significant reason is that knowledge distillation is useful to transfer the learned classification experience from existing teacher models to the BDT, helping the BDT to find the optimal node branching in time, and thus reducing the training time and unnecessary node branching.

To further demonstrate the effectiveness of knowledge distillation, we also show the ΔRules and ΔAccuracy of the BDT in Fig. 11. "Δ" indicates the values with knowledge distillation minus the values without knowledge distillation. As demonstrated, knowledge distillation reduces the classification rules while improving the accuracy. Take Fig. 11b (with the teacher of GRU) as an example, $\frac{\Delta Rules}{\Delta Accuracy} = \frac{-4858}{3.08} = -1577$. That is, when improving every percentage of accuracy, we also reduce 1577 rules.

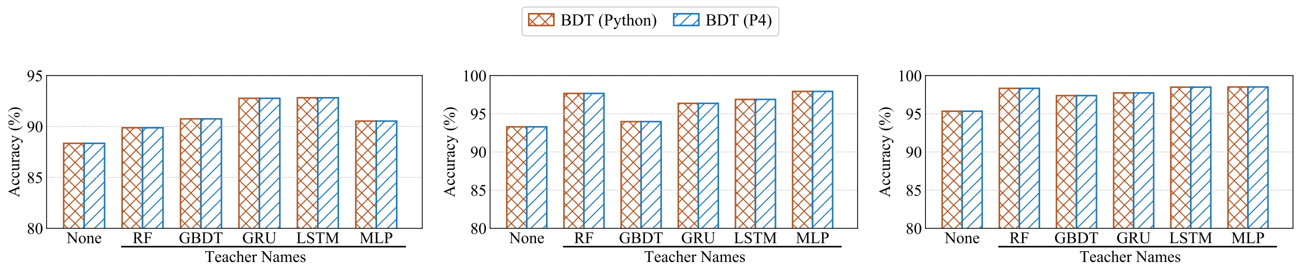
D. Mousikav2 in Switches

The previous experiments are performed with Python. We now further evaluate our scheme on three commodity P4 switches. We first consider the correctness of converting the BDT into ternary match entries. Fig. 12 illustrates the classification accuracies of the BDT based on Python and P4 (in the H3C switch). As depicted, on all three tasks, after encoding the BDT into the P4 program, there is no accuracy difference.

Then, for each evaluated networking task, we deploy the BDT with the best accuracy on the three commodity switches. Fig. 13 shows the switch throughput under the traffic speed of 100Gbps (generated by the KEYSIGHT generator). As shown, after loading the P4 program and installing encoded entries from the BDT, the switch throughput is almost unchanged. Moreover, Fig. 14 shows the packet processing latency of different switches. The latency results show that after deploying the BDT for in-network classification tasks, the latency of switches only increases slightly ($5 \sim 26$ nanoseconds). For example, in Fig. 14a, the latency of the empty switch (without any classification tasks) is 642 nanoseconds. After loading the BDT for the traffic type classification, the latency is 667 nanoseconds, increasing negligibly. That is, the packet processing is still at the line rate after loading the BDT.

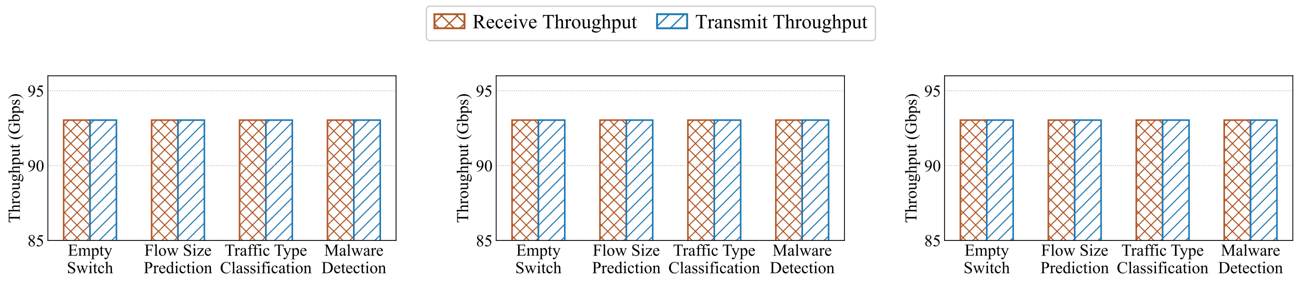
E. Compare With In-Network Solutions

We now compare Mousikav2 with IIsy [9], Planter [13], and pForest [11]. In our setting, IIsy and Planter respectively



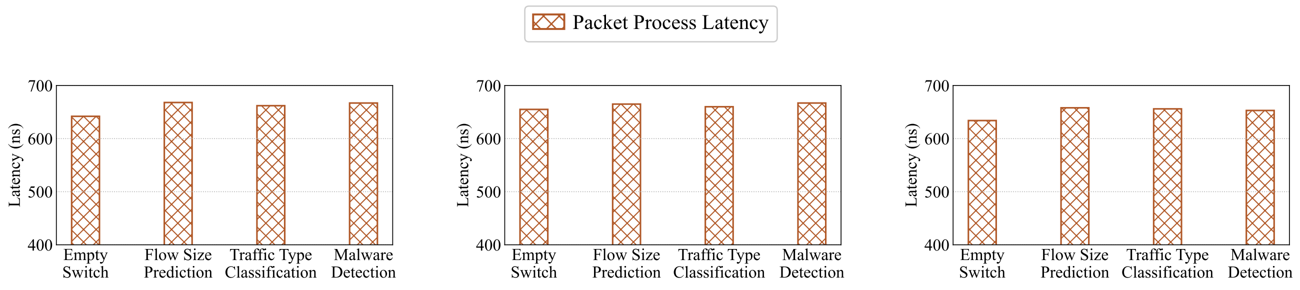
(a) Flow Size Prediction (b) Traffic Type Classification (c) Malware Detection

Fig. 12. The classification accuracy of different BDT versions (Python and P4) on three networking tasks.



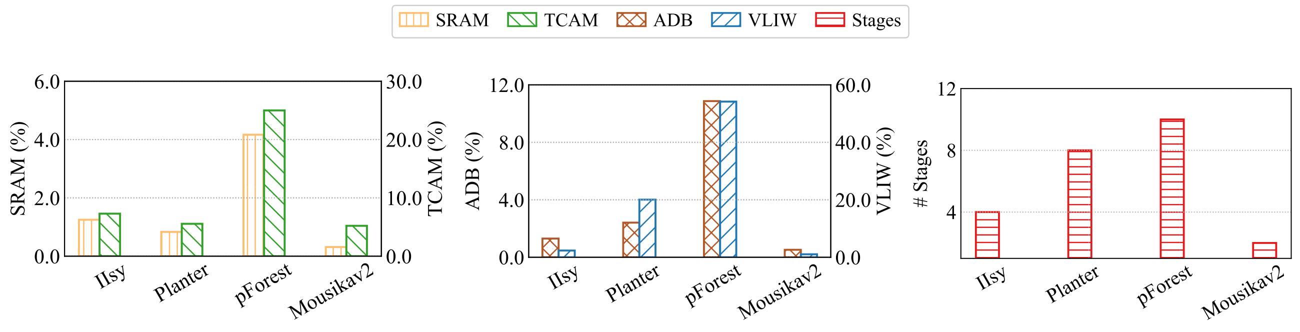
(a) EdgeCore (b) H3C (c) OpenMesh

Fig. 13. The throughput of switches on different tasks. Notably, due to the simulation features (e.g., inter-frame gap and mixed packet sizes), it is impossible to generate traffic of exactly 100Gbps by the KEYSIGHT generator.



(a) EdgeCore (b) H3C (c) OpenMesh

Fig. 14. The latency (in nanoseconds) of three commodity switches before and after deploying the BDT.



(a) Memory Resources (b) Action Resources (c) Number of Stages

Fig. 15. The H3C resource consumption of in-network schemes on the malware detection.

use the feature-decision and direct-mapping manner to deploy the DT. pForest uses the level-table manner to deploy the RF (both the DT and the RF are from Section VII-B). Mousikav2 utilizes the BDT distilled from the RF.

We compare the following switch resources: 1) Memory resources, i.e., the percentage of the used TCAM and SRAM; 2) Action resources, i.e., the percentage of the used action data bus (ADB) and very long instruction word (VLIW); 3) The number of the used switch stages. Fig. 15 depicts the consumption of the mentioned resources on the task of malware detection. As noted, Mousikav2 has the lowest resource consumption. For instance, in Fig. 15a, the TCAM usage of IIsy and Mousikav2 is respectively 7.29% and 5.20% (i.e., 28.67% ↓). In Fig. 15c, Mousikav2 uses only 2 stages. While IIsy leverages 4 stages, which is 2.00× more than Mousikav2. Among these solutions, pForest consumes the most resources (e.g., 25.00% TCAM and 54.17% VLIW) as it encodes several subtrees (i.e., a forest) instead of one DT/BDT in the rest schemes. Although levels of different subtrees can be embedded in one stage to maintain more subtrees, due

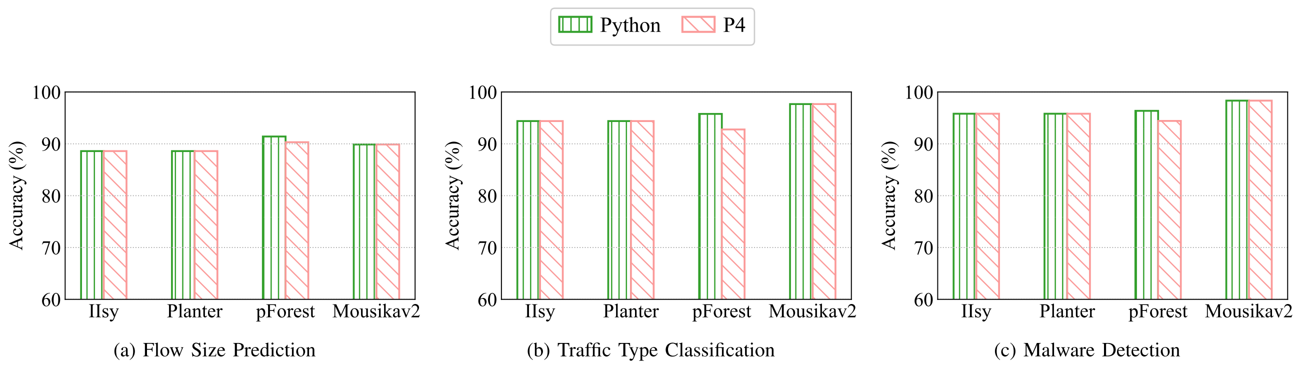


Fig. 16. The classification accuracy of different in-network solutions (Python and P4) on three networking tasks.

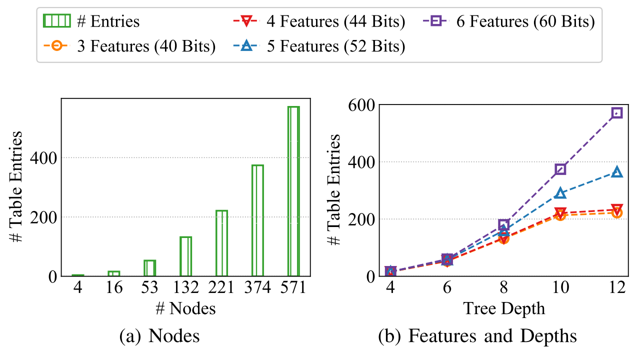


Fig. 17. Size scalability of the BDT.

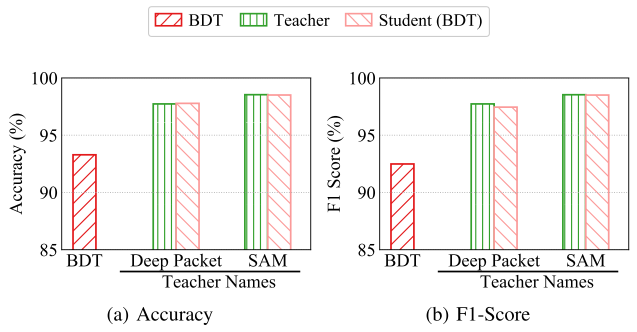


Fig. 18. The performance of the BDT and two traditional NNs on the traffic type classification.

to the resource limitation, pForest can only encode 4 out of 10 subtrees of the used RF in our experiments.

Fig. 16 illustrates the classification accuracy of these in-network solutions. The accuracies of IIsy and Planter are consistent with the DT in Python. As mentioned, pForest cannot deploy the whole RF because of the resource limitation, thus it has a lower accuracy than the RF in Python. Generally, Mousikav2 has a better accuracy than these solutions, e.g., 97.66% vs. 94.39% (IIsy) vs. 94.39% (Planter) vs. 92.77% (pForest) in Fig. 16b.

Although we are able to encode the BDT in one P4 table/stage to demonstrate superiority over other schemes, the encoded BDT may not always consume such a small fraction of resources. Fig. 17 depicts the relationship between the table entries and factors like tree nodes/features/depths of the BDT. As revealed, the number of table entries is positively correlated with these factors. That is, if a task contains too many features or trains a bigger BDT on node/depth, the needed table entries may exceed the memory of one stage. This can be mitigated by feature selection [58] or tree pruning [59], [60].

F. Compare With Traditional NNs

In Section II-A, we introduce several traditional NNs deployed on servers for networking classification. Fig. 18 compares our BDT with two traditional schemes: Deep Packet [4] and SAM [16]. As shown, after the knowledge distillation from traditional schemes, the BDT also has a competitive accuracy. However, Fig. 18b illustrates that the distilled BDT is still weaker than its teachers, e.g., 97.45% vs. 97.73% (Deep Packet) on F1-score.

Therefore, to further boost the performance of the BDT, one may run Mousikav2 in a hybrid fashion like [9]. For example, traffic is first classified by the BDT on switches in line rate, and only part of the traffic (with low classification confidence or needing a fain-grained classification) is redirected to the server for an intensive NN-based investigation. In this way, we can reduce the latency and load of server-based works while improving the overall classification accuracy/F1-score.

VIII. Conclusion and Further Discussion

In this paper, we present Mousikav2 to tackle the drawbacks of offloading the ML models to the switch. We redesign the DT algorithm, getting the binary decision tree (BDT). The classification rules of the BDT are of bits, which can be encoded by the well-supported ternary match in a new proposed resource-efficient P4 program (consuming only two stages). Also, given the complexity of other ML models, in Mousikav2, we adopt a teacher-student knowledge distillation architecture to transfer other models to the unified BDT. By doing so, we can not only utilize their classification knowledge for better performance, but also avoid their sophisticated deployments in switches. Mousikav2 yields superior performance in comprehensive experiments. On the traffic type classification, the BDT after knowledge distillation improves the accuracy of the DT by 3.27%. Meanwhile, compared with IIsy [8], Mousikav2 reduces the memory (TCAM) consumption by 28.67%.

Nonetheless, Mousikav2 also has limitations. First, the input features of the BDT are simply parsed to bits, which ignore the bit dependence within a feature. Hence, it may weaken the feature representations when compared with features of integer/floating-point format. E.g., in Fig. 5, the BDT without knowledge distillation is of slightly lower accuracy than the DT. Second, though the current BDT is logically encoded into one table (and thus one stage), the BDT size may increment, e.g., requiring more features or nodes in some tasks, which results in the table entries overflowing the memory of several stages. One may only use the top important features [58] or prune the tree [59], [60] to reduce the model size. Third, knowledge distillation may not always work. Evidence in [53] and [54] that some sophisticated models can generate students of lower accuracy. Given the model diversity, one may have to examine several teacher models per task so that a desired BDT of better accuracy is generated. We will optimize these problems in the future.

Acknowledgment

The authors thank the editors and anonymous reviewers for helping them improve the paper.

References

[1] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Netw.*, vol. 32, no. 2, pp. 92–99, Mar. 2018.

[2] J. Li and Z. Pan, "Network traffic classification based on deep learning," *Trans. Internet Inf. Syst.*, vol. 14, no. 11, pp. 4246–4267, 2020.

[3] P. Poupart et al., "Online flow size prediction for improved network routing," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–6.

[4] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Soft Comput.*, vol. 24, no. 3, pp. 1999–2012, Feb. 2020.

[5] S. M. Kasongo and Y. Sun, "A deep learning method with wrapper based feature extraction for wireless intrusion detection system," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101752.

[6] Z. Zhao, X. Shi, Z. Wang, Q. Li, H. Zhang, and X. Yin, "Efficient and accurate flow record collection with HashFlow," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1069–1083, May 2022.

[7] P. Bosshart et al., "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[8] Z. Xiong and N. Zilberman, "Do switches dream of machine learning?: Toward in-network classification," in *Proc. 18th ACM Workshop Hot Topics Netw.*, Nov. 2019, pp. 25–33.

[9] C. Zheng et al., "Iisy: Practical in-network classification," 2022, *arXiv:2205.08243*.

[10] J.-H. Lee and K. Singh, "SwitchTree: In-network computing and traffic analyses with random forests," *Neural Comput. Appl.*, pp. 1–12, Nov. 2020. [Online]. Available: https://link.springer.com/article/10.1007/s00521-020-05440-2

[11] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-network inference with random forests," 2022, *arXiv:1909.05680*.

[12] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *Proc. SIGCOMM Poster Demo Sessions*, Aug. 2021, pp. 12–14.

[13] C. Zheng et al., "Automating in-network machine learning," 2022.

[14] P4 Language Consortium. *v1model.p4*. Website. Accessed: Sep. 13, 2022. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4

[15] Barefoot Networks. *Tofino Switch*. Website. Accessed: Sep. 13, 2022. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/

[16] G. Xie, Q. Li, and Y. Jiang, "Self-attentive deep learning method for online traffic classification and its interpretability," *Comput. Netw.*, vol. 196, Sep. 2021, Art. no. 108267.

[17] A. Aleesa, M. Younis, A. A. Mohammed, and N. Sahar, "Deep-intrusion detection system with enhanced UNSW-NB15 dataset based on deep learning techniques," *J. Eng. Sci. Technol.*, vol. 16, pp. 711–727, Feb. 2021.

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

[19] P4 Language Consortium. *Core.p4*. Website. Accessed: Apr. 27, 2023. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/core.p4#L72

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 26th Annu. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[22] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[23] H. Peng et al., "Large-scale hierarchical text classification with recursively regularized deep graph-CNN," in *Proc. World Wide Web Conf. World Wide Web (WWW)*, 2018, pp. 1063–1072.

[24] X. Liu et al., "Attention-based bidirectional GRU networks for efficient HTTPS traffic classification," *Inf. Sci.*, vol. 541, pp. 297–315, Dec. 2020.

[25] T. Pan et al., "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 194–206.

[26] D. Firestone et al., "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement.*, 2018, pp. 51–66.

[27] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "FlowLens: Enabling efficient flow classification for ML-based network security applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[28] H. Liu, "Efficient mapping of range classifier into ternary-CAM," in *Proc. 10th Symp. High Perform. Interconnects*, Aug. 2002, pp. 95–100.

[29] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-efficient TCAMs for forwarding engines," in *Proc. IEEE INFOCOM 22nd Annu. Joint Conf. IEEE Comput. Commun. Societies*, Mar. 2003, pp. 42–52.

[30] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.

[31] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification Regression Trees*. Belmont, CA, USA: Wadsworth, 1984.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[33] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*.

[34] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[35] S. Kumar. *Essential Guide to Perform Feature Binning Using a Decision Tree Model*. Website. Accessed: May 3, 2023. [Online]. Available: https://towardsdatascience.com/essential-guide-to-perform-feature-binning-using-a-decision-tree-model-90bcc66d61f9

[36] R. Pramoditha. *One-Hot Encode Scalar-Value Labels for Deep Learning Models*. Website. Accessed: May 3, 2023. [Online]. Available: https://medium.com/data-science-365/one-hot-encode-scalar-value-labels-for-deep-learning-models-4d4053f185c5

[37] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *Int. J. Comput. Vis.*, vol. 129, no. 6, pp. 1789–1819, Jun. 2021.

[38] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531s*.

[39] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2014, pp. 2654–2662.

[40] G. Urban et al., "Do deep convolutional nets really need to be deep and convolutional?" in *Proc. 5th Int. Conf. Learn. Represent.*, 2017, pp. 1–13.

[41] J. Bai, Y. Li, J. Li, X. Yang, Y. Jiang, and S.-T. Xia, "Multinomial random forest," *Pattern Recognit.*, vol. 122, Feb. 2022, Art. no. 108331.

[42] N. Frosst and G. E. Hinton, "Distilling a neural network into a soft decision tree," in *Proc. 1st Int. Workshop Comprehensibility Explanation AI ML Co-Located With 16th Int. Conf. Italian Assoc. Artif. Intell.*, vol. 2071, 2017, pp. 1–8.

[43] J. Bai, Y. Li, J. Li, Y. Jiang, and S. Xia, "Rectified decision trees: Towards interpretability, compression and empirical soundness," 2020, *arXiv:1903.05965*.

[44] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.

[45] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, “Characterization of encrypted and VPN traffic using time-related features,” in *Proc. 2nd Int. Conf. Inf. Syst. Secur. Privacy*, 2016, pp. 407–414.

[46] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, “Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset,” *Future Gener. Comput. Syst.*, vol. 100, pp. 779–796, Nov. 2019.

[47] R. Bolla, R. Bruschi, C. Lombardo, and D. Suino, “Evaluating the energy-awareness of future internet devices,” in *Proc. IEEE 12th Int. Conf. High Perform. Switching Routing*, Jul. 2011, pp. 36–43.

[48] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, Oct. 2001.

[49] L. Van Efferen and A. M. T. Ali-Eldin, “A multi-layer perceptron approach for flow-based anomaly detection,” in *Proc. Int. Symp. Netw., Comput. Commun.* IEEE, 2017, pp. 1–6.

[50] F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. 10, pp. 2825–2830, Jul. 2017.

[51] A. Paszke et al., “Pytorch: An imperative style, high-performance deep learning library,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[52] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, “Toward effective mobile encrypted traffic classification through deep learning,” *Neurocomputing*, vol. 409, pp. 306–315, Oct. 2020.

[53] Y. Zhu et al., “Teach less, learn more: On the undistillable classes in knowledge distillation,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2022, pp. 1–14.

[54] C. Wang, S. Zhang, S. Song, and G. Huang, “Learn from the past: Experience ensemble knowledge distillation,” in *Proc. 26th Int. Conf. Pattern Recognit. (ICPR)*, Aug. 2022, pp. 4736–4743.

[55] J. Yim, D. Joo, J. Bae, and J. Kim, “A gift from knowledge distillation: Fast optimization, network minimization and transfer learning,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7130–7138.

[56] W. Park, D. Kim, Y. Lu, and M. Cho, “Relational knowledge distillation,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 3962–3971.

[57] G. Ji and Z. Zhu, “Knowledge distillation in wide neural networks: Risk bound, data efficiency and imperfect teacher,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2020, pp. 1–16.

[58] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, “An efficient design of intelligent network data plane,” in *Proc. 32nd USENIX Secur. Symp.*, 2022, pp. 2461–2478.

[59] J. R. Quinlan, “Simplifying decision trees,” *Int. J. Hum.-Comput. Stud.*, vol. 51, no. 2, pp. 497–510, Aug. 1999.

[60] M. Bohanec and I. Bratko, “Trading accuracy for simplicity in decision trees,” *Mach. Learn.*, vol. 15, no. 3, pp. 223–250, Jun. 1994.

Guorui Xie received the B.S. degree from Sun Yat-sen University, Guangzhou, China, in 2019. He is currently pursuing the Ph.D. degree in computer science and technology with the Tsinghua Shenzhen International Graduate School, Tsinghua University. His main research interests include computer networks, including in-network intelligence, programmable switch development, and networking classification tasks based on machine learning.

Qing Li (Member, IEEE) received the B.S. degree in computer science and technology from the Dalian University of Technology, Dalian, China, in 2008, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2013. He is currently an Associate Researcher with the Peng Cheng Laboratory, China. His research interests include reliable and scalable routing of the internet, software-defined networks, network function virtualization, in-network caching/computing, and intelligent self-running networks.

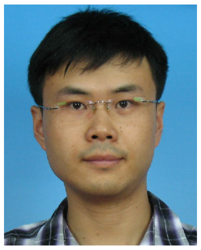
Guanglin Duan received the B.S. degree from Tsinghua University, Beijing, China, in 2020. He is currently pursuing the M.S. degree with the Tsinghua Shenzhen International Graduate School. His research interests include programmable data planes and software-defined programmable network security.

Jiaye Lin received the B.E. degree from the School of Intelligent Systems Engineering, Sun Yat-sen University, Guangzhou, China, in 2022. He is currently pursuing the M.E. degree with the Tsinghua Shenzhen International Graduate School, Tsinghua University, Shenzhen, China. His research interests include in-network intelligence, P4 program development, and reinforcement learning.

Yutao Dong received the B.S. degree from Jilin University, Changchun, China, in 2020. He is currently pursuing the M.S. degree with the Tsinghua Shenzhen International Graduate School. His research interests include anomaly detection in the network environment and AI for networks.

Yong Jiang (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and technology from Tsinghua University, Beijing, China, in 1998 and 2002, respectively. He is currently a Full Professor with the Tsinghua Shenzhen International Graduate School, Tsinghua University. His research interests include future network architecture, internet QoS, software-defined networks, and network function virtualization.

Dan Zhao received the bachelor’s degree in telecommunications from the Beijing University of Posts and Telecommunications in 2011 and the Ph.D. degree in electronic engineering from the Queen Mary University of London in 2015. She was a Post-Doctoral Researcher with the School of Electronic Engineering, Dublin City University, and the School of Computing, National College of Ireland. She is currently an Assistant Researcher with the Peng Cheng Laboratory, Shenzhen, China.



Yuan Yang received the B.Sc., M.Sc., and Ph.D. degrees from Tsinghua University. He was a Visiting Ph.D. Student with The Hong Kong Polytechnic University. He is currently an Assistant Researcher with the Department of Computer Science and Technology, Tsinghua University. His major research interests include computer network architecture and routing protocols.