Programmable Switches for in-Networking Classification

Bruno Missi Xavier*, Rafael Silva Guimarães*, Giovanni Comarela† and Magnos Martinello†
*Federal Institute of Espírito Santo, Espírito Santo, Brazil
†Federal University of Espírito Santo, Espírito Santo, Brazil
Email: {bruno.xavier, rafaelg}@ifes.edu.br, {gc, magnos}@inf.ufes.br

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488840

***Abstract*—Deploying accurate machine learning algorithms into a high-throughput networking environment is a challenging task. On the one hand, machine learning has proved itself useful for traffic classification in many contexts (e.g., intrusion detection, application classification, and early heavy hitter identification). On the other hand, most of the work in the area is related to post-processing (i.e., training and testing are performed offline on previously collected samples) or to scenarios where the traffic has to leave the data plane to be classified (i.e., high latency). In this work, we tackle the problem of creating simple and reasonably accurate machine learning models that can be deployed into the data plane in a way that performance degradation is acceptable. To that purpose, we introduce a framework and discuss issues related to the translation of simple models, for handling individual packets or flows, into the P4 language. We validate our framework with an intrusion detection use case and by deploying a single decision tree into a Netronome SmartNIC (Agilio CX 2x10GbE). Our results show that high-accuracy is achievable (above 95%) with minor performance degradation, even for a large number of flows.**

***Index Terms*—machine learning; software-defined network; P4; in-networking**

I. Introduction

In the recent years, we have seen an increasing interest in research applying Machine Learning (ML) techniques to networking problems [1]. On the one hand, it has been motivated by technological advances in networking, such as network programmability via Software-Defined Networking (SDN) [2]. On the other hand, recent advances in ML have made these techniques flexible and resilient to make them applicable to various real-world scenarios. While programmable switches have been proven to be useful for in-network computing [3], machine learning within programmable switches has had little success so far [4].

With the rise of in-networking computing, the interest in running ML within network devices is rapidly growing for multiple reasons. Firstly, switches offer very high performance. The latency through a switch is in the order of nanoseconds per packet [5], while high-end ML accelerators operate at the scale of tens of microseconds to milliseconds per inference [6]. Also, there are programmable devices such as smart NICs (e.g. Netronome [7], Cavium [8] and Mellanox [9]) that can be placed at servers for accelerating ML use-cases.

Another important motivation to push ML to programmable devices is that the performance of distributed ML is bounded by time required to get data to and from nodes. Therefore, if a switch can classify at the same rate that it carries packets to nodes in a distributed system, then it will equal or outperform any single node [4]. In practice, if a smartNIC is deployed at the edge, the devices allow to terminate data early, reducing the load on the network and improving user experience thanks to reduced latency [10].

One of the challenges to implement ML algorithms within network devices is the hardware implementation complexity required to support mathematical operations. Operations such as addition, xor or bit shifting are feasible, but multiplication, polynomials or logarithms are not pipelined well. However, the RMT architecture [11] has a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. It allows lookups in memory (SRAM and TCAM), manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations using ALUs. We believe that with this new generation of programmable hardware, it is worth rising a question: *can we claim that programmable switches do in-network classification?*

In this paper, our focus is on deploying ML classification trained models showing how to express them into the existing P4 language primitives. More specifically, our contributions are:

- we introduce an innovative framework that enables the transformation of decision-tree models into P4 language pipeline; and
- as proof-of-concept, we build an in-network classifier for an IDS[1] (Intrusion Detection System). We validate our implementation by using the BMv2 emulator [12] and by deploying it into a Netronome SmartNIC.

We conducted extensive experiments in order to assess the models' quality and efficiency. Our evaluation results have shown that high-accuracy for traffic classification is achievable (above 95%) with minor performance degradation. We demonstrate a clear trade-off between accuracy and efficiency when choosing a per-packet or per-flow model. Classifying a flow leads to more accurate results, at the cost of keeping tables' state, and it can be performed by observing only a few of its packets, i.e., when the flow is still "young". In contrast, the per-packet model is stateless having lower accuracy but

[1]Our source code is publicly available at https://github.com/nerds-ufes/in-network_ml

suffering from the issue of flow fragmentation (i.e., different labels for packets in the same flow).

The remaining of this work is organized as follows. Section II presents the framework's architecture and implementation decisions. In Section III, we show how our methodology can be used to support intrusion detection. We position our work in the literature in Section IV and in Section V, we present concluding remarks and directions for future work.

II. Programmable Data Plane as a Classification Machine

The goal of this section is to present the necessary steps to deploy a machine learning model into a programmable networking device by using the P4 language [2]. To do so, the "usual" machine learning process has to be rethought once we want a fast and accurate classifier, under the limitations of the P4 language expressiveness and of the target hardware resources (e.g., memory and processing power). Therefore, the success of our proposal depends on approaching the networking traffic classification problem at the right granularity (e.g., packets or flows) and choosing P4-suitable features and models.

Our framework can be seen as a shift from the standard match-action paradigm. Traditionally, switches' actions are driven by table matching, which can range from using only MAC addresses to more complex OpenFlow [13] rules. Now, P4 enables us to express the matching as the result of mathematical computations over features that can also be extracted from packet header and computed using e.g. stateful registers. We claim that such computations can be obtained from simple learning algorithms so that they can be helpful in different networking scenarios. One advantage of expressing the matching as a result of a learning algorithm is the possibility of having networking data-driven updates. In other words, if the nature of the traffic changes, new models can be built and deployed into the data plane.

Figure 1 shows the high-level architecture of our framework and its main components. The architecture is composed of three planes:

1) The *Knowledge Plane* performs the transformation of knowledge into the a machine learning model. In this context, knowledge may refer to information obtained from external datasets, In-band Network Telemetry, or active measurements. The *Feature Extractor* (Section II-A) component is in charge of selecting and transforming the knowledge base's fields into useful features, considering the limitations of the P4 language. The *Machine Learning Model Builder* (Section II-B) component trains the model based on the features that coming from the previous component.
2) The *Control Plane* connects the Knowledge and Data planes, and it has the goal of mapping the previously trained model to an application, given a target hardware architecture. When the *ML to P4 Compiler* (Section II-C) component receives a model from the *Machine Learning Model Builder* component, it creates a new application code based on the P4 language, and/or a new set of match-action rules. Then, the *Firmware builder* component uses the application code to compile the firmware for a specific target architecture and sends it to the *Agent Deployer* component, which, in turn, installs the new firmware into the physical device. Meanwhile, match-action rules, can be sent by the *Controller* to the data plane via a southbound interface.
3) In the *Data Plane*, when the switch receives the new firmware and the set of match-action rules from the *Control Plane*, it assumes a new behavior previously determined by the machine learning model. At this point, the switch recognizes the class that the packet or flow belongs to and reacts immediately to execute either forwarding or dropping policies.

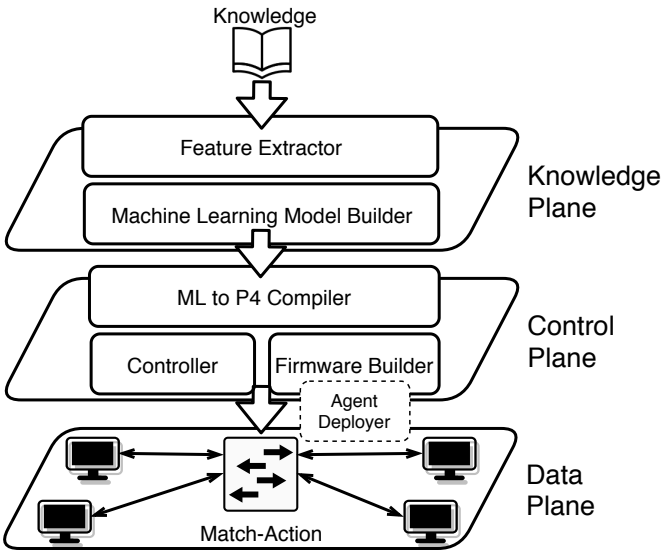

Fig. 1. Framework's Architecture

The following sections present details and considerations about the three most important components of our framework: *Feature Extractor*, *Machine Learning Model Builder*, and *ML to P4 Compiler*.

A. Feature Extractor

A crucial step in machine learning is to precisely identify the objects of study and the characteristics, i.e., features, that define those objects. Traditionally, networking traffic data focus on two objects (or granularities), *flows* or *packets*, and features are extracted from headers and payloads. The right choice of granularity and features heavily depends on the application.

There is a trade-off when choosing between a per-packet or a per-flow model, especially in our context, where the goal is to build high-accuracy models capable of processing packets at line-rate. On the one hand, per-flow features can be computed by aggregating information from several packets, and by exploring these richer features, it is possible to train better models than when using only packets. On the other

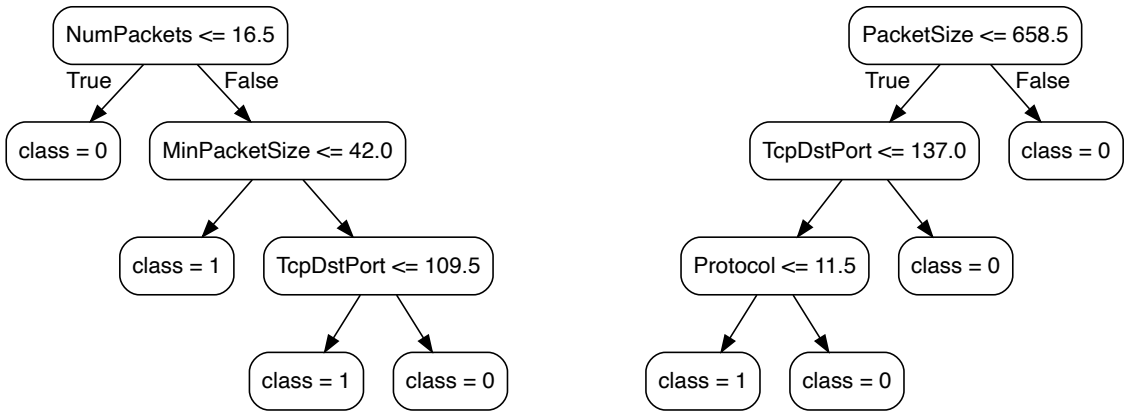

Fig. 2. Examples of decision trees for classifying flows (left) and packets (right).

hand, in order to classify a flow, it is necessary to keep up-to-date flows' states in memory, which can be resource intensive. Hence, a per-packet model has the potential to be more efficient, but it does not offer to the model aggregated measurements and the possibility of learning from temporal correlation.

Once the proper granularity is defined, in general, as many features as possible are extracted from the data. Then, feature selection and/or dimensionality reduction techniques are applied in order to restrict the set of candidates to a new set, with the more adequate and/or informative ones. Unfortunately, the processes of extracting and applying transformations to features commonly rely on operations (or primitives) that can not be easily (or efficiently) expressed in P4, our target language.

A canonical example of this issue is the lack of floating-point operations in P4. This characteristic of the language (and in many cases of the underlying hardware as well) disallows us to compute standard metrics (e.g., average and standard deviation) and to apply important data transformations (e.g., logarithm). Therefore, one must be aware of these limitations when building a model. Otherwise, it will not be possible to translate the model in question to a P4 application, and the task of in-network classification will not be achieved. In Section III, we discuss how we overcame this issue when instantiating our framework for the scenario of intrusion detection.

B. Machine Learning Model Builder

Once the dataset is ready, the next step is to build the classification model. The machine learning literature has a plethora of supervised learning techniques, but not all of them are suitable for our task in hand. More specifically, in addition to the usual requirements of a ML model (e.g., accuracy and generalization), here we are interested in building a model, f, that satisfies the following properties when classifying an instance $\mathbf{x}$:

1) the necessary operations to compute $f(\mathbf{x})$ must be readily available in P4; and
2) computing $f(\mathbf{x})$ must be fast, given a target hardware.

For our framework, we decided to use a decision tree in an attempt to satisfy these two properties. It is out of the scope of this work to go into the details of many algorithms to justify our choice. Even so, we provide the main reasons for not choosing the following popular options: k-Nearest Neighbors (k-NN), Naive Bayes (NB), Support Vector Machine (SVM), and Artificial Neural Network (ANN). The interested reader can find more about these methods in the machine learning literature (e.g., [14]–[17]).

The k-Nearest Neighbors is an instance-based learning algorithm, which means that the model is the training data itself. In addition, sophisticated data-structures are necessary to find the closest elements to a given $\mathbf{x}$ that we wish to classify. Hence, if the target hardware has limited memory and processing power, it is unlikely that the classification operation will satisfy Property (2).

In order to classify an instance $\mathbf{x}$, the Naive Bayes classifier yields the class that maximizes the conditional probability of observing a certain class given $\mathbf{x}$. It is known to be a fast classifier, which means that it satisfies Property (2). However, P4 lacks the exponentiation operation which is necessary in some variations of algorithm, thus, violating Property (1). Even if using a NB variation that can be expressed in P4, we opted for not doing so because the algorithm is known to give poor classification results when the features in $\mathbf{x}$ violate the (naive) assumption of conditional independence.

There are two main versions of the SVM classifier: with a linear or with a non-linear kernel (e.g., Radial Basis Function – RBF). The former satisfies Properties (1) and (2), but it is only appropriate for linearly-separable problems. The latter is able to deal with more complex problems, but it may violate Property (1), if the kernel demands certain operations (e.g.,

```
...

table classtable {

    key = {
        meta.class: exact;
    }
    actions = {
        forward_by_class;
        ...
    }
    size = 512;
    default_action = NoAction;
}

...

apply {
    // extract and aggregate features from the
        packet's headers. The extracted features are
         hdr.ipv4.totalLen and hdr.tcp.dstPort
    extract_features();

    // compute a per-flow identifier by hashing (
        source IP, source Port, destination IP,
        destination Port, IPv4 Protocol)
    hash();

    // update the per-flow features according to the
         packet's features and flow identifier. It
        also retrieves the flow's updated features
    update_features();

    // Apply the decision tree model to the flow
    if (meta.NumPackets <= 16.5)
        meta.class = 0;
    else
        if (meta.MinPacketSize <= 42.0)
            meta.class = 1;
        else
            if (meta.TcpDstPort <= 109.5)
                meta.class = 1;
            else
                meta.class = 0;

    classtable.apply();
}

...
```

Listing 1. Main portions of the P4 code obtained by applying the *ML to P4 Compiler* component to the per-flow tree of Figure 2.

```
...

table classtable {

    key = {
        meta.class: exact;
    }
    actions = {
        forward_by_class;
        ...
    }
    size = 512;
    default_action = NoAction;
}

...

apply {

    // Apply the decision tree model to the the
        packet. In this case, features are extracted
         directly from the packet's headers
    if (hdr.ipv4.totalLen <= 658.5)
        if (hdr.tcp.dstPort <= 137.0)
            if (hdr.ipv4.protocol <= 11.5)
                meta.class = 1;
            else
                meta.class = 0;
        else
            meta.class = 0;
    else
        meta.class = 0;

    classtable.apply();
}

...
```

Listing 2. Main portions of the P4 code obtained by applying the *ML to P4 Compiler* component to the per-packet tree of Figure 2.

exponentiation), and Property (2), due to the need of storing all support vectors and executing many kernel computations for classifying a single $\mathbf{x}$.

Similarly to a SVM with a non-linear kernel, classifying a given $\mathbf{x}$ with an ANN is challenging in our scenario. The classification can be costly, for involving matrix-vector multiplication operations, and it may depend on activation functions that are not available in P4.

We do not claim that it is not feasible to optimize/adapt/approximate the methods aforementioned to perform in-network classification, but we do argue that a decision tree classifier is a more natural choice for such a task given the P4 language current primitives. In order to classify an element $\mathbf{x}$ with a decision tree model, only comparisons are necessary, and those can be easily expressed in P4 through *if-else* statements (more details in Section II-C).

For instance, suppose that we are interested in a model for classifying packets. Figure 2 shows an example of a decision tree model for three features (*PacketSize*, *TcpDstPort* and *Protocol*) and two classes (*class 0* and *class 1*). When a packet needs to be classified (right of Figure 2), the first step is to extract the necessary features, i.e., packet size, IPv4 protocol, and TCP destination port, from its headers; the second step is to use these values to traverse the tree, from root to leaf, respecting the conditions stated in each tree-node; and finally, the last step is to yield the reached leaf's label.

In general, decision tree models are capable of separating non-linear problem, but in some cases, this classifier may not be as powerful as other more complex approaches (e.g., a neural network with several hidden layers). If during the training phase one realizes that a decision tree is not an appropriate choice, then our framework can be easily extended to use a Random Forest model instead. The classification time will grow on the number of trees, but it is also expected to yield more accurate results.

C. ML to P4 Compiler

Classifying an element $\mathbf{x}$ with a decision tree involves traversing the tree from root to leaf, respecting the conditions in each node until reaching a leaf. When a leaf is reached, its label is returned. Traditionally, in general-purpose programming languages, this process can be easily implemented by using recursion or repetition loops. However, neither of these options are available in the P4 language. Hence, an alternative is to hard-code the tests and labels within the tree-nodes into *if* and *else* statements.

To that end, the *ML to P4 Compiler* component implements a recursive algorithm responsible for traversing the previously trained decision tree in pre-order fashion. The algorithm adds to the P4 code the statement `if (<condition>)` if the node being processed is not a leaf. Then, the left subtree represents the path to be followed when `<condition>` evaluates to *true*, while the right subtree adds an `else` to the code, representing the path to be followed when `<condition>` evaluates to *false*. A leaf node holds the class to which it belongs, and when this node is reached, a new entry in the code updates the state variable of the current classification.

The *if-else* chain generated by *ML to P4 Compiler* component is added into a template of P4 program that describes the generic behavior proposed for a given P4 application. These generic functionalities are not affected by the result of the ML to P4 compilation and can be written regardless of the result of the component's execution (e.g., *parser* and *deparser* control, actions and tables). For instance, the per-flow model presented in Figure 2 (left) is transformed, by the *Compiler* component, into the *if-else* chain contained in Listing 1. This Listing also shows how every packet traversing the device is processed. Whenever a new packet arrives, the relevant features from its headers are identified and saved into the pipeline's metadata. These stored features are accounted for the composition of new cumulative features, such as total packets and total bytes from the flow. Then, a hash identifier that maps the packet to its flow is computed. This hash is calculated based on the source and destination IP addresses, source and destination ports, and the IPv4 Protocol field from the packet's header. Before the classification, the flow's features are updated in the SRAM memory registers. The actions *extract_features* and *update_features* are written into the P4 code by the *Compiler* component, adding just the features used by the decision tree model. On the other hand, the *hash* action does not require any evaluation by the compiler since the hash key is generated with the features that identify a flow, regardless of the ML model. After updating the flow's features, a classification occurs in the following *if-else* chain. Eventually, the flow's class will be stored in the *meta.class* variable. Next, the table *classtable* drives the packet to the match-action rules defined by the *Controller* component, according to each class and the forwarding or dropping policies.

Similarly to Listing 1, Listing 2 presents the result of applying the *ML to P4 Compiler* component to the per-packet decision tree of Figure 2. When a new packet is received in the pipeline, the relevant features are identified. Differently from the per-flow case, there is no need to compute hash functions or update flow's features. In fact, the packet's features are accessed directly from its headers. When the *if-else* chain is finalized, the packet's class is stored in the *meta.class* variable. Finally, the pipeline is then directed to apply the *classtable* match-action rules defined by the *Controller* component according to each class and the forwarding or dropping policies.

TABLE I
DATASET SUMMARY

Class	Number of flows	Number of packets
Benign (BE)	436,183	12,260,490
DOS GoldenEye (GE)	7,574	66,795
DOS Hulk (HK)	14,108	1,245,906
DOS Slowhttptest (SH)	4,218	32,510
DOS Slowloris (SL)	3,894	37,236
Web Brute-force (BF)	1,356	19,755
Port Scan (PS)	67,579	174,312

III. CASE STUDY: INTRUSION DETECTION

In this section, we present a use case for validation of our framework with a scenario of IDS (*Intrusion Detection System*). More specifically, we show that a simple machine learning model (a single decision tree) can be deployed into a SmartNIC to accurately detect different types of attacks (Section III-D) with acceptable latency degradation (Section III-E). We describe the dataset that we used, the challenges involved, and our experimental setup in Sections III-A, III-B, and III-C, respectively.

A. Dataset

To conduct our experiments, we relied on the dataset created and made available by Sharafaldin *et al.* [18]. The dataset is composed of PCAP files related to network traffic generated in five days (Monday to Friday), whose flows were labeled as being benign or a specific type of attack. The authors argue that they addressed many of the issues related to older, and possibly outdated, datasets. Moreover, they show that machine learning can be used to accurately distinguish the types of flows by applying well-known algorithms and carefully-designed feature extraction/selection strategies. It is not our goal to dig into the details about the characteristics of each type of flow, how they were generated, or how they were used for classification purposes, and we refer the interested reader to the aforementioned manuscript for more information.

From the available data, we selected the PCAP files from two days (Wednesday and Thursday), which contain flows related to seven types (or classes), one *benign* (BE) and six attacks: *DOS GoldenEye* (GE), *DOS Hulk* (HK), *DOS Slowhttptest* (SH), *DOS Slowloris* (SL), *Web Brute-force* (BF), and *Port Scan* (PS). There are other types of attacks in these two days, but we decided to remove them from our analysis due to extremely low frequency. A brief summary of the dataset that we are using is presented in Table I.

B. Challenges

A natural step to deploy a machine learning model to perform intrusion detection into the data plane would be using the features and models studied in [18]. To that end, one could simply use the P4 language to write code to compute the features and to perform the classification. Unfortunately, there are issues related to software, hardware, and the nature of the application itself that prevent such a direct approach. The most important ones are enumerated below:

1) P4 is not a general-purpose programming language. Therefore, many of the features used in [18] can not be directly (and maybe not efficiently) computed (or adapted) using P4.
2) From a practical point of view, classifying a flow after it ends is not a useful task, because the intrusion may have already happened. An in-network IDS must be able to accurately identify a malicious flow as soon as possible to prevent harm.
3) Classifying flows requires keeping per-flow table entries to store their respective features values. Moreover, these entries must be updated after every packet belonging to the flow passes through the hardware where the model is deployed. Therefore, it is necessary to understand whether the overhead of performing a table lookup, entry update, and flow classification is not prohibitive in a high-throughput environment.
4) If instead of classifying flows, due to the cost of the classification, one decides to classify packets individually, then it is not necessary to maintain per-flow table entries. In other words, in a per-packet scenario, the overhead related to table lookup, entry update, and memory usage is not present anymore. However, a single packet may be considerably less informative than a flow with regard to the classification task. Hence, it is necessary to understand the trade-off between per-packet and per-flow models.

Next, we present our implementation decisions and experimental methodology to address the four challenges listed above.

C. Experimental setup

In order to compare the per-packet and per-flow strategies, we decided to create two models, one trained with features from individual packets and another with features from flows. Table II shows the features used when building each model. One advantage of working with flows is the possibility of having features which can capture their dynamics (e.g., *duration* and *cumulative packet size* at a given moment) which cannot be accomplished by relying only on packets. One disadvantage of working with flows is the impossibility of directly computing better descriptive measures (e.g., *average*, *variance*, and *standard deviation*) for the time-varying features, since important operations, such as division and square root, are not supported in P4. In the per-packet case, our choice of features was motivated by [4]. In fact, we used the same ones, except the IPv6-related features (once the dataset in use only contains IPv4 traffic) and TCP/UDP source port (once they are, in general, randomly chosen by the operating system [19]).

TABLE II
PER-FLOW AND PER-PACKET FEATURES

Model	Feature	Short description
Per-flow	EtherType	From Ethernet header
	Protocol	From IPv4 header
	CumIPv4Flag_X	Cumulative number of occurrences of the IPv4 flag X, for each $X \in \{\text{DF}, \text{MF}\}$
	TcpDstPort	TCP destination port
	UdpDstPort	UDP destination port
	CumPacketSize	Cumulative sum of IPv4 packet size
	FlowDuration	Time interval since the arrival of the first packet
	MaxPacketSize	Size of the largest packet
	MinPacketSize	Size of the smallest packet
	NumPackets	Number of packets
	CumTcpFlag_X	Cumulative number of occurrences of the TCP flag X, for each $X \in \{\text{FIN}, \text{SYN}, \text{RST}, \text{PSH}, \text{ACK}, \text{URG}\}$
Per-packet	EtherType	From Ethernet header
	Protocol	From IPv4 header
	IPv4Flag_X	Indicates whether IPv4 flag X is set, for each $X \in \{\text{DF}, \text{MF}\}$
	TcpDstPort	TCP destination port
	UdpDstPort	UDP destination port
	PacketSize	IPv4 packet size
	TcpFlag_X	Indicates whether the TCP flag X is set, for each $X \in \{\text{FIN}, \text{SYN}, \text{RST}, \text{PSH}, \text{ACK}, \text{URG}\}$

In our next step, we extracted two samples from the dataset, one for training and the other for testing the models. To that end, we randomly sampled 100 thousand flows for the training set and another 100 thousand flows for the test set. In both cases, the proportions of flows in each class were set to be the same as in the whole dataset.

Although we used the same training set for the per-flow and per-packet models, the feature extraction process differs significantly in these two cases. In the per-packet case, we simply extracted the features described in Table II from five, randomly selected, packets from each flow. The per-flow case is more complicated, once features must be extracted from flows, not packets. For each flow in the training set, we updated the values of the features in Table II after processing each packet of the flow in the same order as they appear in the original PCAP files. Then, we took five equally-spaced samples from the sequence of features states (we used the percentiles 20%, 40%, 60%, 80% and 100% as the observation points to ensure that each flow was sampled at different stages).

To create the models, we relied on Python's *scikit-learn*[2] implementation of the decision tree classifier, and we used cross-validation to choose the best hyperparameters (e.g., tree height and minimum number of items per leaf) in order to avoid overfitting.

Finally, we conducted two experiments to test the models. First, we applied the per-packet model to all packets in the test set, and the per-flow model to all flows (each flow was

[2]https://scikit-learn.org/

TABLE III
SUMMARY OF CLASSIFICATION RESULTS

	Per-flow - After Last Packet			Per-flow - After First Packet			Per-packet		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Benign (BE)	0.97	0.99	0.98	0.98	0.97	0.98	1.00	0.98	0.99
DOS GoldenEye (GE)	1.00	0.87	0.93	0.00	0.00	0.00	-	0.00	-
DOS Hulk (HK)	1.00	1.00	1.00	-	0.00	-	0.78	1.00	0.87
DOS Slowhttptest (SH)	0.87	0.94	0.90	-	0.00	-	0.84	0.15	0.25
DOS Slowloris (SL)	0.99	0.95	0.97	-	0.00	-	0.97	0.63	0.77
Web Brute-force (BF)	0.69	0.95	0.80	0.04	1.00	0.08	0.61	0.18	0.28
Port Scan (PS)	1.00	0.94	0.97	1.00	0.94	0.97	1.00	0.71	0.83
All Attacks combined	0.99	0.96	0.97	0.99	0.96	0.97	0.86	0.96	0.91

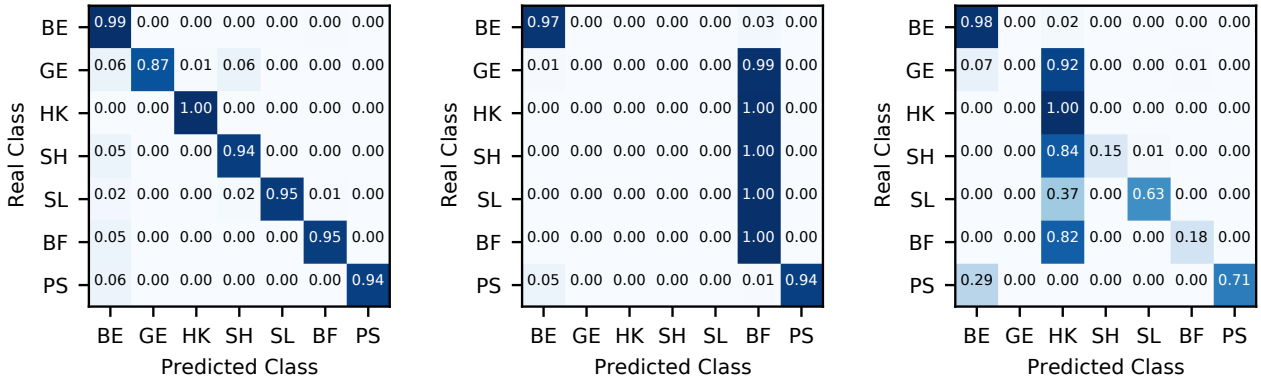

Fig. 3. Confusion matrices for: per-flow classification after the last (left) and the first (middle) packets of each flow are observed; and per-packet classification (right). Values are normalized by the sum of each row.

classified once for each one of its packets). This experiment was performed using a Python script and a BMv2 emulator [12], and it aimed at assessing the quality of the models (according to several classification metrics) and observing the behavior of the P4 code in a controlled environment.

Our second experiment was designed to understand the overhead added to standard packet processing in order to execute the operations necessary to the classification task (feature extraction, updating features values, and classifying the flow). To that end, we deployed each model into a Netronome[3] SmartNIC (Agilio CX2x10GbE), connected via PCI Express 3.0 to an Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz processor and 16GB of RAM. This SmartNIC has two physical and four virtual interfaces. In this experiment, we randomly selected 100 thousand packets and we sent these packets, one at a time, to the virtual interface `vf0_0`. Then, inside the SmartNIC, each packet was handled according to four versions of our P4 application:

1) The first version simply forwards the packet to the virtual interface `vf0_1`. The purpose of this version is to serve as baseline.
2) The second version contains the per-packet model. It extracts the per-packet features listed in Table II, classifies the packet, and then forwards it to the virtual interface `vf0_1`.
3) The third version contains the per-flow model. It extracts the per-flow features from the packet, updates the features values related to the flow containing the packet, and forwards the packet to the virtual interface `vf0_1`.
4) Finally, the fourth version is similar to the third one, but it also classifies the flow, with the per-flow model, before forwarding the packet to the virtual interface `vf0_1`.

In all these cases, we computed the time difference between the instant immediately after the packet enters the virtual interface `vf0_0` and the instant right before the packet leaves the SmartNIC via virtual interface `vf0_1`.

D. Classification Results

Table III and Figure 3 summarize the classification results for our per-flow and per-packet models, including the *confusion matrices*, *precision*, *recall*, and *F1-Score*. We used the per-flow model in two scenarios: for classifying each flow after it finishes (i.e., after its *last* packet is observed) and to classify each flow after it is just born (i.e., after its *first* packet is observed). The former is not useful in practice, but it gives an upper bound with regard to the quality that our models can achieve. Similarly, the latter gives us a lower bound. Moreover, the last row of Table III contains the results for the binary version of the problem, where the model is used to distinguish between *benign* and *attack* (regardless of the type of attack).

On the one hand, it is possible to observe that classifying a flow after it finishes yields high F1-score values. In fact, these

[3]https://www.netronome.com/products/agilio-cx/

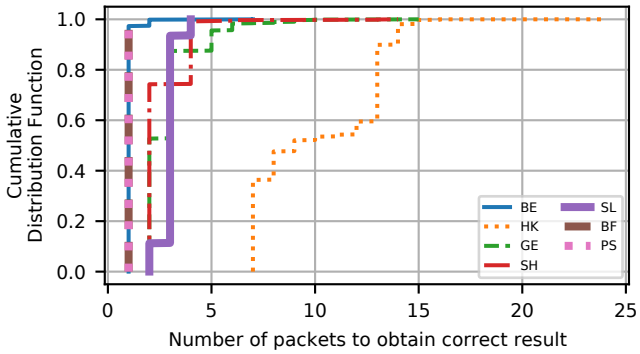

Fig. 4. Empirical cumulative distributions for number of observed packets within a flow until correct classification result is obtained with the per-flow model. We only computed for flows that were correctly classified after observing its last packet.

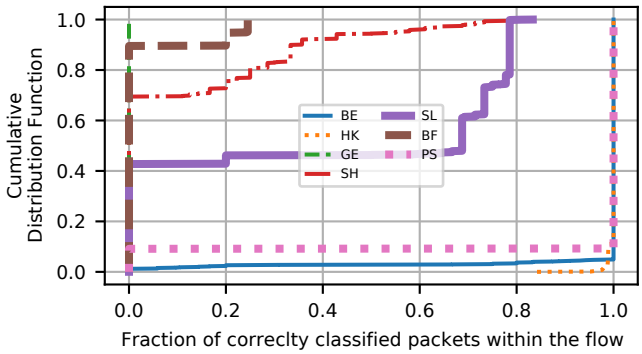

Fig. 5. Empirical cumulative distribution for the fraction of correctly classified packets within the same flow according to the per-packet model.

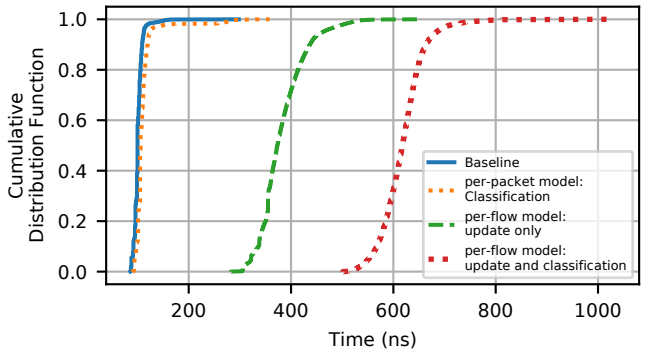

Fig. 6. Empirical cumulative distribution for the time that a packet takes to traverse (in and out) the whole pipeline at the SmartNIC in different scenarios.

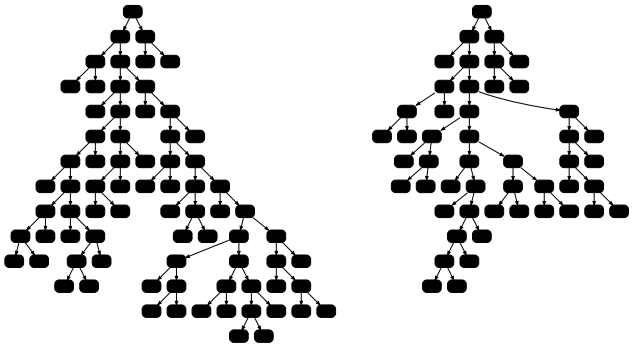

Fig. 7. Structure of each decision tree model: (left) per-flow and (right) per-packet.

numbers are close to the results obtained in [18], when a larger number of (more complex) features was available. On the other hand, the same is not true when the classification is performed right after a new flow appears. The *benign* class still has high F1-score, but most of the attacks are incorrectly classified as *Web Brute-force*. These numbers may suggest that classifying a "young" flow is a nearly-infeasible task, but it is important to emphasize that if one is interested only in distinguishing between *benign* and *attack*, then a 0.97 F1-score is achieved.

Motivated by the possibility of classifying "young" flows, we computed the number of packets that need to be observed in each flow in order to obtain the correct classification result for the first time (given that such correct result is eventually obtained). As shown in Figure 4, with the exception of the class *DOS GoldenEye*, with five packets, most of the flows are properly labeled. Moreover, for some classes, including *benign*, observing one or two packets is enough.

The per-packet model has similar results to the second per-flow scenario (after observing only the first packet). In other words, the per-packet model is not capable of distinguishing different types of attacks, but it yields reasonable results in the *benign vs. attack* problem, although a considerable fraction of malicious packets are labeled as *benign*.

One issue that can arise with a packet-based model is the *fragmentation* of flows. Suppose that a packet of a legit TCP flow is incorrectly labeled as malicious. This scenario may lead to unexpected (and potentially disastrous) behaviour if, for instance, the network is careless programmed to immediately drop such a packet. Hence, it is important to understand whether our per-packet model classifies packets from the same flow differently. According to Figure 5, fragmentation does arise in many cases. However, it is important to notice that in the *benign* class, around 97% of the flows did not suffer from the fragmentation issue.

E. Performance Evaluation Results

Our last set of results concerns the cost of performing in-network flow and packet classification. Figure 6 presents the distributions of the time that a packet resides inside the SmartNIC according to the four versions of our P4 application enumerated in Section III-C. The time that a packet needs to be forwarded from `vf0_0` to `vf0_1` averages around 100ns. The cost added to classify packets, using the per-packet model, is almost negligible. However, a significant overhead arises when we use the per-flow model. The necessary time for extracting the features of a packet and updating its respective flow entry averages around 400ns, and when the classification is performed as well, the total time that takes to process a single packet rises, in average, to about 650ns.

One interesting aspect of Figure 6 is the difference between the costs of the classification operations performed by the per-packet and per-flow models. Since the per-packet classification overhead is negligible, one should expect that the per-flow classification overhead (without the operations of features extraction and flow update) should be negligible as well. In order to dig deeper into this issue, we looked at the difference between the complexities of these two models. If the per-flow model were significantly more complex than the per-packet one, then a significant cost difference could be explained. However, Figure 7 shows that, although there is a difference between the two trees (especially with regard to their heights), it is not as significant as the difference in the overheads. Therefore, there are more intricate reasons (e.g., memory management) responsible for the distinction between the per-flow and per-packet classification costs.

F. Considerations

The results that we described in this section entail a series of important considerations. Perhaps, the most important one is the clear trade-off between accuracy and efficiency when choosing a per-packet or per-flow model. On the one hand, classifying a flow leads to more accurate results, at the cost of keeping a per-flow table. On the other hand, despite having low overhead, the per-packet model has lower accuracy and suffers from the issue of flow fragmentation. In short, choosing one or the other is a decision that can only be made after carefully analyzing the application requirements.

In spite of our high-accuracy results, we do not claim that our decision tree models are ready to be deployed in the wild. First, traffic patterns are always changing. Even though the datasets that we used were carefully generated [18], it is possible that over time they become outdated, and a machine learning model is only as good as the representativeness of the training set. Second, a per-flow model may be memory-intensive since it requires storing flow-related metrics. Eventually, the network device may run out of memory, leading to extreme performance degradation. The solution to both of these problems passes through the interaction between the control and data planes. More specifically, the control plane can assume the role of removing inactive flow-entries from the device's memory and receiving important monitoring information from the data plane. Then, such information can be sent to the knowledge plane in order to help building a better model.

Another important consideration is related to the limitations of our performance evaluation study. We showed that the latency overhead for the per-packet model is negligible, but the per-flow models inflate the forward time in 6.5x. Despite informative, it is also important to understand the impact of our models on the device's memory and throughput in different scenarios.

Finally, it is possible to alleviate the per-flow classification cost significantly. We can use the fact that few packets are necessary to correctly classify a flow. To do so, we can change our P4 program to stop computing statistics about a flow after observing a predetermined number of packets, deleting its respective table entry, and marking the flow, in another table, accordingly to its inferred class. This new table stores less per-flow bytes and it allows the next packets to be properly labeled without the need of updating the flow statistics and applying the classification operation.

We emphasize that all these limitations and considerations are subjects of our ongoing/future work.

IV. Related Work

Recent years have seen an unprecedented surge in research combining ML and networking. For example, programmable network devices can be used to accelerate neural networks processing [20], to improve distributed ML through in-network aggregation [21], deployed at the edge as virtualized ML functions [22], congestion control [23] and distributed reinforcement learning [24]. This paper is complementary to these works, focusing on one specific aspect of ML, classification.

In terms of traffic classification using ML, there has been a set of recent works [25]–[28] [4]. The closest work in the literature to our framework is [4]. Our approach differs from all previous works, by introducing a per-flow model with fined-grained evaluation. Also, we assess the classification results presenting a latency analysis comparing per-flow and per-packet models, while in [4] they have presented high-level experimental results. To the best of our knowledge, our contribution is a first step toward a pragmatic transformation of decision-tree models into P4 language pipeline.

V. Conclusions and Future Work

In this paper, we have introduced a framework for in-network classification addressing the problem of creating simple and reasonably accurate ML models that can be deployed into the programmable data plane with minor performance degradation. We validate our framework with an IDS use case. We implement decision tree models using the BMv2 emulator [12] and deploy them into a Netronome SmartNIC, as a proof-of-concept. Our results show that high-accuracy is achievable (above 95%).

In addition to the specific ongoing/future tasks enumerated in Section III-F, this work motivates a series of broader questions and research directions. First, as in our IDS case study, are there other networking scenarios where flows can be classified appropriately after observing only a few of its packets? Second, given the optimizations previously discussed, is it possible to use per-flow models in more sophisticated devices (e.g., Barefoot Tofino - TNA)? Finally, can we combine our in-network classifier with smart agents that decide when to perform new measurements and train/deploy new models?

If these questions can be positively answered, then the work presented in this manuscript can be seen as part of a broader framework. One that is capable of performing complex networking functions, with little human supervision and intervention.

Acknowledgment

This study was financed by CAPES (Finance Code 001), CNPq, FAPES, CTIC, and RNP.

REFERENCES

[1] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[3] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: Association for Computing Machinery, 2017, p. 150–156. [Online]. Available: https://doi.org/10.1145/3152434.3152461

[4] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, pp. 25–33.

[5] B. Tofino. (2020) Barefoot Tofino. [Online]. Available: https://barefootnetworks.com/technology/#tofino

[6] N. P. Jouppi, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[7] NETRONOME. (2019) Netronome Agilio SmartNIC. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[8] CAVIUM. (2019) Cavium LiquidIO SmartNICs. [Online]. Available: https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/

[9] MELLANOX. (2019) Mellanox Innova Ethernet Adapter. [Online]. Available: http://www.mellanox.com/products/smartnic/

[10] D. R. Mafioletti, C. K. Dominicini, M. Martinello, M. R. N. Ribeiro, and R. d. S. Villaça, "Piaffe: A place-as-you-go in-network framework for flexible embedding of vnfs," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[12] P4 Language Consortium. P4-bmv2 website. [Online]. Available: https://github.com/p4lang/

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, Mar. 2008.

[14] T. M. Mitchell, *Machine Learning*, 1st ed. USA: McGraw-Hill, Inc., 1997.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[16] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

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

[19] M. Larsen and F. Gont, "Recommendations for transport-protocol port randomization," BCP 156, RFC 6056, January, Tech. Rep., 2011.

[20] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, ser. NetCompute '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 20–25. [Online]. Available: https://doi.org/10.1145/3229591.3229594

[21] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," 2019.

[22] R. Souza, C. Trois, R. Turchetti, M. Martinello, J. H. G. Correa, D. R. Mafioletti, L. C. E. Bona, J. C. D. Lima, and A. Machado, "Mlfv: Network-aware orchestration for placing chains of virtualized machine learning functions," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[23] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, 2019, pp. 3050–3059.

[24] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 279–291. [Online]. Available: https://doi.org/10.1145/3307650.3322259

[25] G. Sun, L. Liang, T. Chen, F. Xiao, and F. Lang, "Network traffic classification based on transfer learning," *Computers & electrical engineering*, vol. 69, pp. 920–927, 2018.

[26] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2018.

[27] 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 Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[28] K. L. Dias, M. A. Pongelupe, W. M. Caminhas, and L. de Errico, "An innovative approach for real-time network traffic classification," *Computer Networks*, vol. 158, pp. 143–157, 2019.