UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

BRUNO LOUREIRO COELHO

Detecting DoS Attacks Utilizing Random Forests in Programmable Data Planes

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Alberto Egon Schaeffer-Filho

Porto Alegre
December 2020

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitoria de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Sérgio Luis Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

Nowadays, most services rely on an Internet connection to be accessed by their clients. Ergo, even a brief disruption of connection can cause considerable loss, monetary or otherwise. Therefore, it is important that potential denial of service (DoS) attacks are detected quickly, in order to avoid or minimize the impact they may have on the availability and quality of services. Recent technological advances in programmable networks – specifically the programmability of data planes in switches and routers, have made available new ways of detecting such attacks. Utilizing this newfound possibility, this work proposes the utilization of Random Forests, a Machine Learning technique, to aid in quickly and accurately detecting DoS attacks in a programmable switch. Random forests utilize several procedurally generated classification trees, each of them independently classifying an input as one of a set of classes. Here, each decision tree will classify a network flow as potentially malicious, i.e. part of a DoS attack, or a legitimate user flow. Despite utilizing multiple classification trees to improve accuracy, random forests are relatively light-weight, with each tree requiring few and simple computations to arrive at a classification. The simplicity of the operations executed in each tree makes this technique a good candidate for use in programmable switches, since they have limited resources and require fast processing to operate at line rate.

Keywords: Traffic classification. programmable data planes. random forests. artificial intelligence. DoS attacks. networks. network security.

Detectando ataques DoS utilizando florestas aleatórias em planos de dados programáveis

RESUMO

Hoje em dia, a maioria dos serviços dependem de uma conexão com a Internet para serem acessados pelos seus clientes. Portanto, mesmo breves disrupções de conexão podem causar perdas consideráveis, monetária ou de outros tipos. Assim, é importante que possíveis ataques de negação de serviço (DoS) sejam detectados rapidamente, a fim de evitar ou minimizar o impacto que possam causar à disponibilidade e qualidade de serviços. Avanços tecnológicos recentes em redes programáveis - especificamente a programabilidade de planos de dados em *switches* e roteadores, tornaram disponíveis novas maneiras de detectar este tipo de ataque. Utilizando essas novas possibilidades, este trabalho propõe a utilização de florestas aleatórias, uma técnica de aprendizado de máquina, para ajudar na rápida e precisa detecção de ataques DoS através de *switches* programáveis. Florestas aleatórias utilizam várias árvores de classificação proceduralmente geradas, cada uma independentemente realizando a classificação de uma entrada em um conjunto possível de classes. Neste trabalho, cada árvore de decisão irá classificar um fluxo de dados da rede em potencialmente malicioso, isto é, parte de um ataque DoS, ou em fluxo legítimo. Apesar de utilizar várias árvores de classificação para aumentar a acurácia, florestas aleatórias são relativamente leves, com cada árvore precisando realizar poucas e simples computações para obter uma classificação. A simplicidade das operações executadas em cada árvore fazem essa técnica uma boa candidata para o uso em *switches* programáveis, uma vez que estes possuem recursos limitados e requerem processamento rápido para operar em taxa de linha.

Palavras-chave: classificação de tráfego, planos de dados programáveis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

DP	Data Plane
RF	Random Forest
AI	Artificial Intelligence
ML	Machine Learning
NN	Neural Network
CV	Computer Vision
RL	Reinforcement Learning
SVM	Support Vector Machine
DoS	Denial of Service
DDoS	Distributed Denial of Service
ISP	Internet Service Provider
IPv4	Internet Protocol version 4
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
FSM	Finite-State Machine
DPI	Deep Packet Inspection
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Unit
IoT	Internet-of-Things
IAT	Inter-Arrival-Time
EWMA	Exponentially Weighted Moving Average

CONTENTS

1 INTRODUCTION

This work focuses on the detection of malicious attacks that may affect computer networks. In particular, we utilize a popular Machine Learning (ML) technique, Random Forest (RF), to efficiently detect Denial of Service (DoS) attacks in the data plane of a programmable switch. In Section 1.1, we provide further contextualization on these issues, while in subsequent sections we present the motivation (Section 1.2) and goals of this work (Section 1.3), along with the organization of the remaining of this document (Section 1.4).

1.1 Contextualization

Presently, many services are offered online, having their customers access them through the Internet. As such, networks have become an increasingly important infrastructure for services, seeing as it is often relied upon (KUROSE; ROSS, 2012). Not only is it important to have Internet access, but also that it is always available, as even brief disruptions of connection can impact these services, potentially leading to economic losses (WOLFE, 2018). Along with appropriate equipment and planning of infrastructure, another concern of both Internet Service Providers (ISPs) and network administrators is the possibility of a malicious attack.

In order to try to affect online services, perpetrators may use a group of hosts infected with malware to attempt to slow or sever the access of legitimate users to the victim system (KUROSE; ROSS, 2012). These attacks looking to compromise the quality or availability of services are known as Denial of Service (DoS) attacks, with a variant where the attack originates from multiple sources being known as Distributed Denial of Service (DDoS) attacks (ZARGAR; JOSHI; TIPPER, 2013). These attacks have been constantly increasing in number and size, with volumes of over 300 Gbps being reported in 2013 (DONG; ABBAS; JAIN, 2019), while more recently a DDoS attack generated more than 1.35 Tbps (1350 Gbps) in 2018 (AKAMAI, 2018).

Considering the importance of avoiding service downtime, it is imperative to quickly and accurately detect potential attacks. However, that is not an easy task, as attacks have become more sophisticated, making it harder to differentiate between legitimate and malicious traffic (MOORE et al., 2006). Detection can be done by different means, with broad classes of techniques being packet-based and flow-based analysis.

In packet-based analysis, one of the most common techniques is Deep Packet Inspection (DPI), where the contents of each packet are carefully examined, as opposed to only the headers of the protocols being used (ANTONELLO et al., 2012). This analysis can have as objective the verification of format correctness, validity, or even determining whether it is a malicious or legitimate packet (FINSTERBUSCH et al., 2013). However, this technique has several drawbacks: first, it has a very high processing cost, becoming very difficult for high traffic networks; second, it becomes extremely complex and inefficient when we are dealing with encrypted traffic, which has become very common (ILIYASU; DENG, 2019); third, it raises several concerns about net neutrality and privacy (JORDAN, 2009). On those accounts, it is not as often used in the context of real-time DoS detection.

On the other hand, flow-based analysis tries to classify a group of packets belonging to the same connection as potentially malicious or not. In order to do this, classifiers may calculate statistical values (NOSSENSON; POLACHECK, 2015) such as the average, maximum, and minimum packet size or the time in between the arrival of consecutive packets of a specific flow. However, even with these values available, it is not trivial to determine the legitimacy of network traffic, considering modern attacks have become better at having their flows appear seemingly harmless.

Recent breakthroughs in the field of programmable networks have allowed for further programmability of switches and routers. As opposed to common devices that come with a set of functionalities implemented by their manufacturers, programmable switches allow researchers to propose and evaluate new ideas in sufficiently realistic settings. As these devices are in the data path of both malicious flows and legitimate user flows, performing the detection of DoS attacks in their programmable data planes is an interesting alternative to traditional approaches.

1.2 Motivation

Current techniques for DoS detection generally lack in at least one of accuracy, detection speed, or scalability (ZARGAR; JOSHI; TIPPER, 2013). This can be partially attributed to systems being forced to decide between utilizing the limited but efficient interfaces provided by traditional switches or utilizing more software-based approaches, allowing more customization at the cost of efficiency (YAN et al., 2015).

With the advent of programmable data planes, however, we aim to create a system capable of efficiently processing individual packets in order to classify their respective flows into likely being part of an attack or genuine user traffic. To achieve this, we utilize the P4 language (BOSSHART et al., 2014) to insert a pre-trained model of Random Forest (HO, 1995) into the programmable switch, calculating statistical values and utilizing the RF to accurately classify flows at line rate.

1.3 Goals

The goal of this work is to define, implement, and evaluate a system that combines state-of-the-art techniques in artificial intelligence, namely Random Forest, with recently developed programmable switches attending to the P4 language specification. Once concluded, we aim to have achieved a system that can not only accurately classify malicious DoS flows, but can also do it at line rate.

In order to reach this goal we aim to complete the following sub-goals:

- Study the related work, focusing on implementations of ML algorithms and DoS detection techniques in programmable data planes;
- Perform classification of network flows in the data plane;
- Train a Random Forest classifier with appropriate datasets;
- Evaluate our implementation with realistic datasets.

1.4 Outline

The remaining of this work is organized as follows. In Chapter 2, we present background on *DoS attacks*, highlighting the characteristics of different types of attacks; *programmable data planes*, the technology utilized in this work; and *random forests*, the artificial intelligence technique that is employed in our proposed system. In Chapter 3, we discuss related work, focusing on DoS detection and relevant work on programmable data planes. In Chapter 4, the proposed system is more thoroughly described, explaining how several of its aspects were implemented. In Chapter 5, we show the conducted experiments, relating the employed methodology and the obtained results. In Chapter 6, we present a conclusion to this work, suggesting possible future work in order to improve the system proposed in this work.

2 BACKGROUND

In this chapter, we present theoretical background on three topics: DoS attacks (Section 2.1), as it is important to understand the differences between types of attacks, in view of the fact that their characteristics may influence which method is more appropriate for the detection; programmable data planes (Section 2.2), once we must know the capabilities and limitations of the technology utilized; and Random Forests (Section 2.3), in order to provide justification for employing the selected model in this work.

2.1 Denial of Service

Denial of Service is a broad class of attacks that share a common objective to hamper or deny access from legitimate users to a target online service or server. A large subclass of DoS attacks is Distributed Denial of Service attacks, which as the name implies, originate from more than one source. While there are many types of DoS attacks with varying characteristics, there is not a single universally agreed upon taxonomy (SHARAFALDIN et al., 2019).

There have been many different propositions when it comes to finding a set of criteria for organizing these types of attacks, such as the degree of automation, scanning strategy, architecture (ASOSHEH; IVAKI, 2008), exploited vulnerability, source address validity, attack rate dynamics (MIRKOVIC; REIHER, 2004), transport protocol (SHARAFALDIN et al., 2019), among others. However, as the taxonomy of the attacks is not the main focus of this work, we will cite a few examples while grouping them into two major categories: high-bandwidth, flooding attacks and low-bandwidth, resource depletion attacks.

In flooding attacks, the perpetrator generates a high volume of packets in order to affect the victim's network. This can be done utilizing protocols such as TCP, UDP, ICMP, and DNS (ZARGAR; JOSHI; TIPPER, 2013), which normally send responses to certain commands. However, the bandwidth required for this type of attack is extremely high, being generally impossible for a single attacker to take down a sizeable server. As such, these attacks tend to be distributed, i.e., DDoS attacks, utilizing several hosts infected with malware, forming what is called a botnet.

While flooding attacks can certainly affect services, for very large services it would take an enormous amount of malicious traffic to take it down. As such, attackers have come up with innovative techniques, aiming to exhaust their target's resources while using very low bandwidth (SIKORA et al., 2020). Some of these attacks, for their low-bandwidth and slow packet sending characteristics, are known as slow DoS attacks. Some examples of tools utilized to create such attacks are Slowloris, GoldenEye, Hulk, and Xerxes (DURAVKIN; LOKTIONOVA; CARLSSON, 2014). Generally speaking, they try to establish many HTTP connections with the target server until its resources are exhausted and legitimate clients cannot connect.

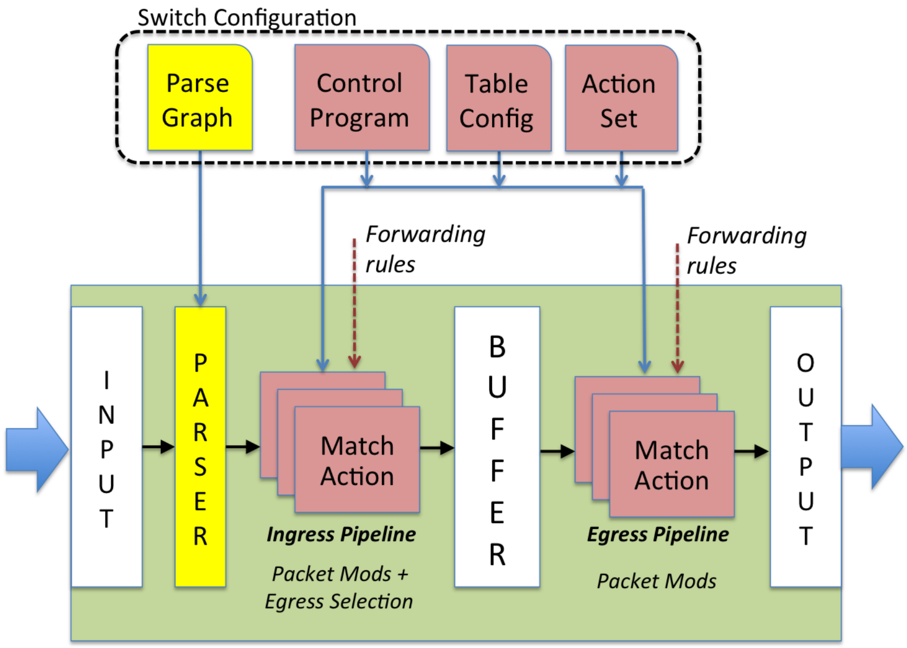
2.2 Programmable Data Planes

Traditional routers and switches come with a set of functionalities, some of which might be optional and allow some degree of configuration. However, for new functionality to be added to them, it requires the manufacturers to design a new device model with new algorithms and then for it to be implemented in hardware efficiently so it can meet the needs of its users. This process can severely slow down the pace of innovation, as it limits the possibility of experimenting with newly proposed ideas (MCKEOWN et al., 2008), such as new routing algorithms, communication protocols, and other network functionalities.

Seeking to facilitate innovation by allowing researchers to reshape the entire process, from creating their own logic in the control plane to reprogramming the data plane, programmable switches were designed. While it is not the only language to allow data plane programmability, the newly proposed P4 (BOSSHART et al., 2014) programming language and interface was designed to be used for programming protocol-independent packet processors (thus the name P4). With it, programmers are relieved of having to know the details of the target hardware by utilizing an abstract forwarding model.

In this abstract forwarding model used by the language, the target device can be viewed as a pipeline with several consecutive stages: a *parser*, responsible for extracting the protocol headers as defined by the programmer; an *ingress* processing block, where usually most of the processing is done, commonly utilizing match+action tables; an *egress* processing block, being generally but not exclusively used in the event of packet cloning; and a *deparser*, where protocol headers are inserted into the forwarded packet. This abstract model can be observed in Figure 2.1, where we see the four stages of the pipeline. Additionally, we can observe some of the objects that compose the switch configuration, such as the parse graph, control program, table configurations, and action set.

Figure 2.1: Abstract Forwarding Model

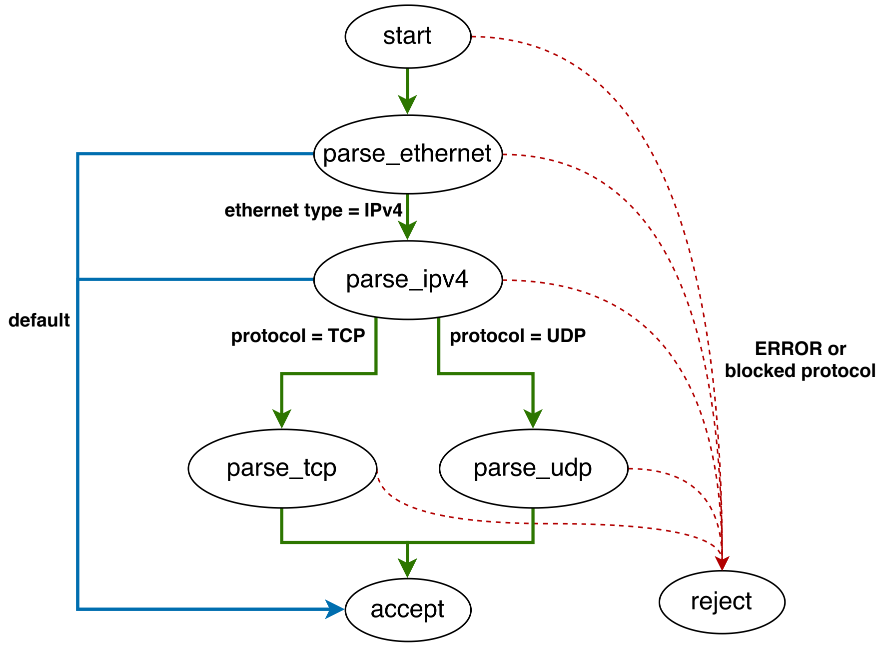


Source: (BOSSHART et al., 2014)

The programmer may define a series of data structures for protocol headers. This includes each header's organization, detailing what fields compose it and each of their size in bits, while also specifying the relative order in which they appear. While these data structures usually have a fixed shape and length, there is also support for extractions with variable lengths, such as is the case with the Transmission Control Protocol (TCP) options header, varying from 0 to 320 bits in size (KUROSE; ROSS, 2012).

The aforementioned header definitions are informed to the switch in the source code, having this compiled and installed in the device before use. Once the device is operational, whenever a packet arrives it is first sent to the parser. There, each header is extracted following the order and logic defined by the programmer (BUDIU; DODD, 2017). This is done by defining states and rules for transitions between them, establishing a Finite State Machine (FSM). An example of an FSM can be seen in Figure 2.2, where we can observe conditional transitions from one state to another. A state corresponds to the extraction of a specific header type, such as an *IPv4 header*, which may be followed by a *UDP header*, a *TCP header*, or none. The final states of the FSM are *accept*, forwarding the packet to the next stage for further processing, and *reject*, marking the packet to be dropped.

Figure 2.2: Example of a Finite State Machine describing the Parser



Source: Author

During the processing of each packet, the programmer may utilize match+action tables. These units, as the name implies, attempt to match the assigned keys to values specified in the entries of the table being applied. On a successful match, the determined action is invoked with the appropriate parameters, according to the corresponding table entry. Although match+action tables are not a novel concept introduced by P4, the language allows programmers to customize exactly what keys and actions to use. Thereby, programmers are no longer required to utilize built-in fields of common headers when matching keys or simple preprogrammed procedures as actions, such as matching the destination address field of an Internet Protocol version 4 (IPv4) header in order to assign a port for the packet to be forwarded through. With P4, it is possible to utilize any key, be it a header field value or a metadata value, along with customized actions, allowing researchers to experiment with different algorithms. A visual example of a match+action table can be seen in Table 2.1. In this example, we observe the representation of a match+action table that defines some rules for IPv4 forwarding. Using the first line as an example, we see that if the IPv4 destination address attribute of a packet matches with 10.0.1.1 (match value) in all 32 bits (match type), we invoke the action IPv4_forward, passing as parameters the Ethernet address 00:00:00:00:01:01 and egress port 1.

When creating their custom actions, programmers may pick from a set of simple arithmetic or logical operations and from a particularly restrictive set of program flow control primitives (LAPOLLI; MARQUES; GASPARY, 2019). Due to the nature

Table 2.1: Representation of an example match+action table

Match Value IPv4 DST Address	**Match Type**	**Action**	**Parameters Ethernet DST Address**	**Port**
10.0.1.1	/32	IPv4_forward	00:00:00:00:01:01	1
10.0.2.2	/32	IPv4_forward	00:00:00:02:02:00	2
10.0.3.1	/32	IPv4_forward	00:00:00:02:02:00	2
10.0.3.0	/24	IPv4_forward	00:00:00:03:03:00	3
10.0.4.0	/24	Drop	-	-

Source: Author

of switches having to process a massive number of packets per second, P4 does not support traditional loop constructs, such as *for* and *while* commands, usually available in most general-purpose programming languages, such as C, Java, and Python. Additionally, seeing as most protocols work with integer representation of bits or simply raw bits, P4 does not support floating-point numbers or operations (BUDIU; DODD, 2017), as they are more complex and slower than operations over integers. Another important limitation of the language is that it does not support division, as it is also a costly and complex operation. These limitations must be observed when designing a system that will run on a programmable switch, often requiring changes of functionalities or creative ways to find workarounds for these characteristics of the language.

In the ingress processing block, the programmer may utilize match+action tables that belong to the ingress (as opposed to match+action tables that belong to the egress), customized actions, and primitives offered by the language in order to specify the desired packet processing behavior. This block operates over the previously extracted headers and any other custom metadata declared by the programmer, along with standard metadata defined by the P4 language. Metadata information does not usually persist between different packets, with a few exceptions such as cloned or recirculated packets. Aside from a few exceptions, the only persistent data between different packets is the match+action rules, installed by the control plane; counters, which have a very specific semantic; meters, which can be utilized for keeping statistics about packets; and registers, which can be read and modified by the program, however requiring very limited device memory. The egress processing block has many similarities to the ingress block, with the few differences of being invoked for each packet in the case of *cloning*, a different set of user-defined *match+action tables*, and a small set of *standard metadata* values that can be exclusively accessed in this block.

2.3 Random Forests

Random Forest is a technique in Machine Learning (ML) that combines several different Classification Trees (CT), a specific type of Decision Trees (DT), in order to obtain more accurate classifications (HO, 1995). In this section, we review the definitions of Classification Trees and Random Forests, while also highlighting some of their characteristics that motivated the use of these particular techniques in this work.

2.3.1 Classification Trees

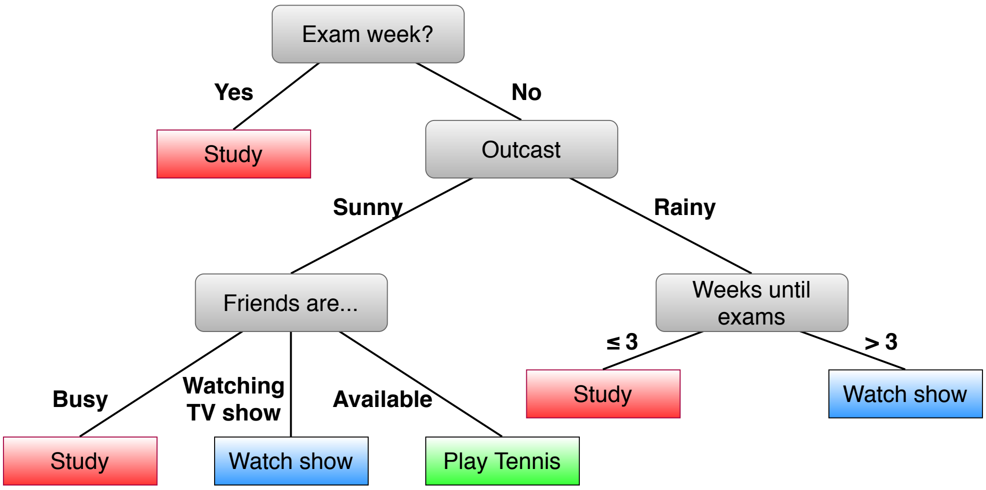
Decision Tree is a technique that performs a series of chained tests or comparisons of values in order to arrive at a decision. In ML, this technique is generally implemented in tree-like data structures, with each node performing a test of a specific parameter against a value, in the case of continuous numeric attributes, or a set of acceptable options in the case of discrete attributes. The algorithm branches to a different node depending on the result of the test, repeating this process until it arrives at a leaf node, where a decision is obtained (QUINLAN, 1987). Classifications Trees are a subset of DT, specifically where the target attribute, that is, the attribute the algorithm is trying to decide on, is a discrete value, possibly indicating a class. A visual example of a Decision Tree can be seen in Figure 2.3, representing decision-making as a series of questions that, depending on their answer, will result in different decisions. As an example, on exam week the process would decide that we should stay in and study. However, on a sunny day outside of exam week, when our friends are available, the process would decide that we should play Tennis.

There are many algorithms that focus on different ways to generate classification trees for a given input set of instances, with a particularly popular one being C4.5 (QUINLAN, 1993). This specific algorithm utilizes the idea of normalized information gain, a value calculated for each attribute of the aggregate training samples. The value of normalized information gain utilizes the entropy of the set of samples, calculated as:

$$S(D) = -\sum_{i \in M} p_i \log_2(p_i) \tag{2.1}$$

where $S(D)$ is the entropy of the set D and p_i is the probability of a sample to belong to class i in the set of possible classes M. Along with the entropy of the whole set D, given the set V of possible values j of the attribute A, we define the entropy of the set after

Figure 2.3: Representation of an example Decision Tree



Source: Author

splitting it into subsets based on the value of the attribute A, generating subsets D_j, as:

$$S_A(D) = \sum_{j \in V} \frac{|D_j|}{|D|} \times S(D_j) \tag{2.2}$$

where $|D_j|$ is the number of samples in the subset D_j and $|D|$ is the number of samples in the set D. Once we have the entropy of the whole set and the subsets calculated, we must first calculate the normalization factor as:

$$SplitS_A(D) = -\sum_{j \in V} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right) \tag{2.3}$$

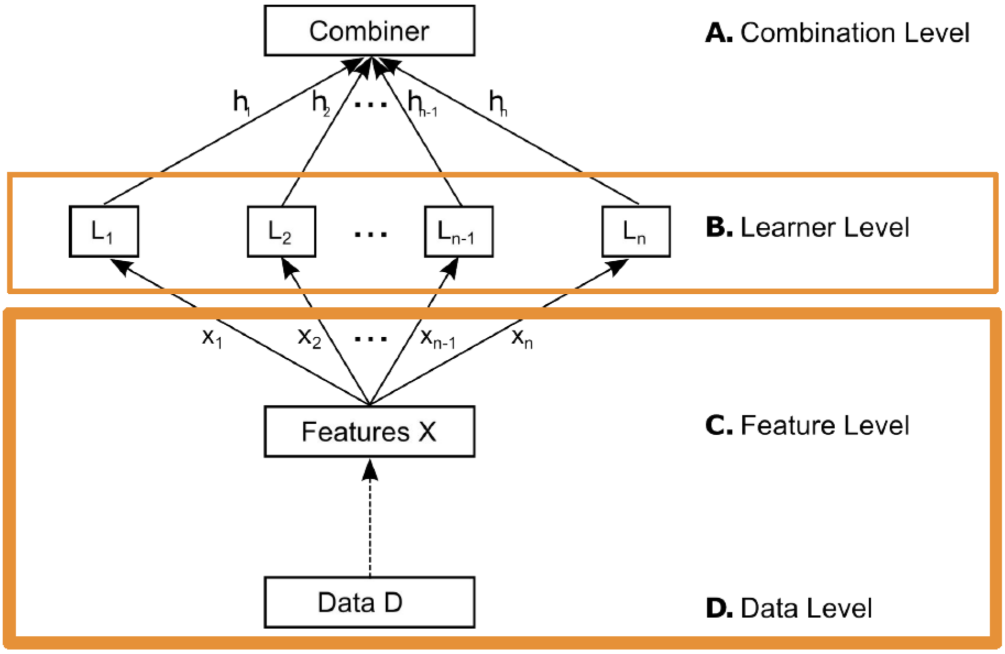
where once again $|D_j|$ is the number of samples in the subset D_j and $|D|$ is the number of samples in the set D. Once this final partial value is calculated, we can finally obtain the normalized information gain by dividing the information gain, defined as the difference of entropy between set and subset, by the normalization factor:

$$NG(A) = \frac{S(D) - S_A(D)}{SplitS_A(D)} \tag{2.4}$$

where $S(D)$ and $S_A(D)$ are the entropy of the dataset D before and after the split by the attribute A, respectively, calculated as shown in Equation 2.1 and Equation 2.2, and $SplitS_A(D)$ is the normalization factor calculated as shown in Equation 2.3. This value is calculated for each attribute A, with the algorithm selecting the attribute with the highest normalized information gain.

When an attribute has a continuous value, a cutoff point can be calculated and two subsets are created, one with values that are lower than this cutoff and the other subset

Figure 2.4: Approaches to building ensemble systems



Source: (KUNCHEVA, 2004)

with values that are equal or greater than the cutoff. Regardless of continuous or discrete attributes, the process of calculating the normalized information gain of each attribute is repeated at each iteration of the algorithm, having one attribute be selected per step. Selecting attributes with the highest normalized information gain first ensures that the attributes that better help predict a class are selected as early as possible in the process, improving accuracy and reducing the size of each tree.

2.3.2 Ensemble

Although Classification Trees may achieve relatively high accuracy over the samples utilized during training, they are prone to suffer from what is called overfitting. This happens when a prediction or classification model adjusts too closely to the sample set used, lacking in capacity to generalize its classifications to new samples that were not present during its training stages. In order to prevent overfitting from happening while also boosting accuracy (BREIMAN, 2001), researchers have proposed several techniques that can be applied during the training of ML models, with a particularly competitive option being ensemble learning (POLIKAR, 2006).

Ensemble-based machine learning systems combine multiple ML models into a single classification system, potentially introducing diversity at different levels of the ensemble system. A proposed set of approaches for building these types of systems defines four different levels: combination, learner, feature, and data levels (KUNCHEVA, 2004). An illustration of this division can be seen in Figure 2.4, where we observe the four aforementioned levels and their order, with data being the lowest level.

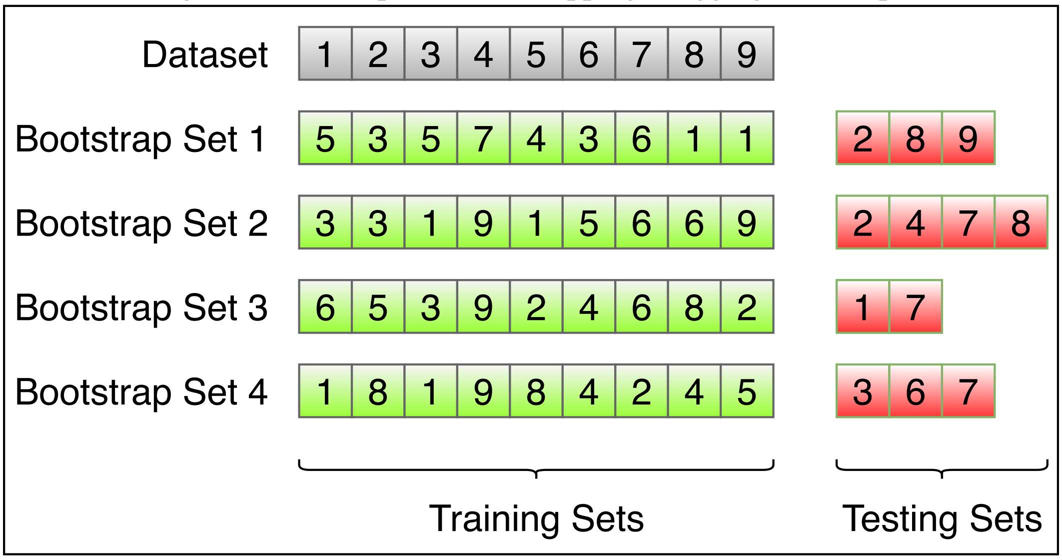
At the combination level, there are different ways to combine the classification obtained by each classifier model into a final classification. The simplest way is a majority vote, where the system counts the number of votes for each classification and picks the classification with the highest number. Other ways include weighted votes, where different classifiers have different weights to their respective votes. In the case of regression trees, where we have a target attribute with a continuous value, we calculate the average or weighted average of the values of each tree.

In the next level, we can have learner diversity, that is, utilizing different models for classifiers. Systems with more than one learner model are called heterogeneous ensemble systems, while the ones with only one learner model are called homogeneous ensemble systems (KUNCHEVA, 2004). This approach can combine many different ML techniques, such as Neural Networks, Classification Trees, Naïve Bayes classifiers, among several other possibilities.

Diversity at the feature level is introduced by selecting a subset of the available features. This subset can be generated based on *natural grouping*, present in some types of features. For features that do not have a natural grouping, an alternative is the *random selection* of a predefined number of attributes, which can be either a ratio over the total number of attributes or a specific value. Other methods of selecting features have been proposed, such as *nonrandom selection* and *genetic algorithms* (KUNCHEVA, 2004).

In the last level, we introduce data diversity, creating subsets of data to be used during training by each learner. The two methods commonly used for this approach are bootstrap aggregating and boosting. In the first method, bootstrap aggregating (also known as bagging), we create several subsets called bootstraps based on the original dataset. Each bootstrap is generated by randomly sampling from the original dataset, allowing duplicates. Usually, it is done by sampling until we obtain a *training set* the size of the original dataset, resulting in a *testing set* with out-of-bag samples, that is, the samples that were left out of the training set. An illustrative example can be seen in Figure 2.5, where we can observe each *training set* having as many samples as the original set, while the size of the *testing set* varies depending on how many samples were not selected in that iteration. The other method utilized in the data level is boosting, which aims to sequentially train

Figure 2.5: Example of Bootstrapping (Bagging) Technique



Source: Author

stronger classifiers by selecting relevant samples (KUNCHEVA, 2004). An improved version of the boosting algorithm is known as AdaBoost, being more commonly utilized than the original version.

Random Forest is an ensemble system that combines several classification trees at the learner level. For the other levels, any of the mentioned techniques can be used, resulting in different forests. For this work, we utilize simple *majority voting* at the combination level, $\sqrt{n}$ features per tree out of the n features available at the feature level, and *bootstrapping*, as previously explained, at the data level.

3 RELATED WORK

In this chapter, we present the related work, highlighting each of the proposed solution's strengths and weaknesses. We focus on related work that proposes techniques for detecting DoS attacks in the data plane (Section 3.1) and related work that presents ways to execute Machine Learning algorithms in the data plane (Section 3.2).

3.1 DoS Detection on the Data Plane

Lapolli, Marques, and Gaspary proposed a system that estimates the entropy of source and destination IP addresses, in order to observe variations in these values (LAPOLLI; MARQUES; GASPARY, 2019). In particular, they observe that in scenarios where a large group of infected hosts, also known as a botnet, targets a particular victim server, the tendency of these entropy values is for the source IP addresses entropy to increase, as more different or spoofed IP addresses are now sending malicious packets, while the destination IP addresses have their entropy decreased, as the botnet will be aiming at a particular target. While this approach works very well, as shown in their experiments, it assumes that there will be a large number of attackers. This can work for certain types of attacks, but the effectiveness for slower, low-bandwidth attacks is unknown. Another difference between this work and our proposed approach is that the system proposed by Lapolli, Marques, and Gaspary can detect the occurrence of attacks, but not classify traffic into malicious or legitimate.

A different approach to detecting DoS attacks was proposed by Febro, Xiao, and Spring. In this work, the authors focus on the detection and classification of DDoS attacks that target SIP in the application layer (FEBRO; XIAO; SPRING, 2019). While the obtained results were positive, the deep packet inspection technique employed in this work focuses on attacks that target this application only. Additionally, a major limitation of DPI is its inability to handle encrypted traffic, which has become very popular. As such, it is not a solution for other types of DoS attacks, whereas our proposed system can be used to detect a wider range of attacks.

Simsek et al. propose an approach for identifying spoofing techniques, in order to pinpoint potential hosts from where DDoS attacks originate (SIMSEK et al., 2020). Their technique aims to detect illegitimate traffic as close to the attackers as possible, in order to drop these packets as soon as possible in the route, preventing the victim from being overwhelmed. While it is an interesting approach, this work focuses on volumetric attacks, that is, attacks with high-bandwidth volume and a massive number of packets. Thus, slower DoS attacks may be undetected by such a system, whereas our proposed system can be used for the detection of this type of attack.

Musumeci et al. utilize and compare several ML classifiers with a programmable data plane in order to detect DoS attacks (MUSUMECI et al., 2020). In the presented solution, the authors extract statistical features from the network traffic and then forward them to an ML module that performs the classification per se. The work compares different ways to realize the first part of the process, with the simplest way being packet mirroring. Other ways include header mirroring and a more advanced metadata extraction done in the data plane. While the extraction can be done in either the controller or the programmable device, the ML module is not implemented in the data plane. This brings about a longer delay before classification, on top of the device not being able to take any appropriate actions with the obtained classification.

pForest (BUSSE-GRAWITZ et al., 2019) implements several RF classifiers in the programmable data plane. This system classifies network flows in order to detect potential DDoS attacks. In addition to stateless features, pForest encodes several stateful features into a single register, trying to minimize the number of bits per feature. For mean values, which are more complex to calculate, they replace the moving average with exponentially weighted moving average, with a weight of $\frac{1}{2}$, which simplifies the computation with the limited capabilities of programmable switches. Despite their positive results, our proposed system utilizes a single RF, diminishing resource usage. Additionally, instead of relying on a fixed set of features for implementing the RF classifier, our work uses an extensible feature library, partially developed within our research group (SAUERESSIG, 2020). Finally, our proposed mechanism better approximates means with an algorithmic approximation of moving averages.

3.2 Machine Learning on the Data Plane

BaNaNa SPLIT (SANVITO; SIRACUSANO; BIFULCO, 2018) is an Artificial Neural Network (NN) Accelerator that utilizes programmable switches to attempt to provide lower latency on online inference of a previously trained NN. Neural Networks, in a simplified way, are several layers of artificial neurons, in which the layer i takes an input, processes it, and generates an output that is then used by neurons in the layer $i + 1$.

While each neuron is usually an extremely simple activator, comparing its weighted inputs against a threshold, the dense structure of NNs can obtain very good results in certain areas, such as Computer Vision (CV). This Accelerator aims to eliminate some of the inefficiencies of usual systems, where after receiving the packet via a network interface, the computer has to send the data from the CPU to the general-purpose GPU (GPGPU) used as an accelerator. While BaNaNa SPLIT does a commendable job as an NN accelerator, we believe NNs are not an appropriate ML model for DoS detection in programmable data planes, as the limited capabilities of programmable switches require several simplifications.

SwitchML (SAPIO et al., 2019) is another system that leverages the capabilities of programmable switches to act as a ML accelerator. This work, however, aims to accelerate not the inference step of ML systems, but the distributed training of ML models. With the tendency of ML models to become bigger and more complex, there are situations in which the use of a single computer to train a system is not feasible. Therefore, a viable option to operate over large ML models is to perform distributed training, in which we have separate workers training different parts of the whole structure. After each iteration, once every worker has finished its workload, the training weights must be synchronized. SwitchML utilizes programmable switches to perform on-path data aggregation, severely reducing network load and delay for weight synchronization, by efficiently and smartly leveraging the physical topology of the network. It might be possible, in future work, to utilize a system like this to perform and accelerate distributed training for DoS detection.

iSwitch (LI et al., 2019), similarly to some of the other related work, aims to be an accelerator for distributed Reinforcement Learning (RL) training. The system can make both synchronous and asynchronous training faster and more efficient. In the case of synchronous training, where workers wait for the updated weights before continuing their computations, iSwitch lowers this wait time, making the training more efficient. While in asynchronous training workers do not wait for weight updates, thus not being blocked during that wait, utilizing outdated weights can still slow down the pace at which the model converges. Therefore, iSwitch can help distributed RL models undergoing asynchronous training to converge faster, both in terms of iterations and time. On top of that, the authors propose a hierarchical distributed training structure to further improve training efficiency. Despite the acceleration provided by this system, RL may not be the most appropriate model for DoS detection, as it must define states and it relies on constant feedback.

IIsy (XIONG; ZILBERMAN, 2019) is a prototype that implements several ML algorithms in a programmable switch. In this work, Decision Trees, Support Vector Machine (SVM), K-means, and Naïve Bayes classifiers are implemented into the programmable data plane. As a testing example, the classifiers are previously trained (offline) to classify Internet-of-Things (IoT) data into one of several categories. The authors focus on the efficient implementation of these different models, aiming to utilize every available feature that P4 enabled switches have to offer. Although the authors implement a DT for network traffic classification, they employ it for a different goal than ours. Additionally, we believe RFs to be a better model for our task, as the slight increase in computational cost provides important refinement of predictions.

N2Net (SIRACUSANO; BIFULCO, 2018) is a proposed system that aims to implement Neural Networks inference in a programmable data plane. The partial work, published in early 2018, implements a prototype system that realizes the forward pass of a binary NN. The authors suggest that their work may be used as a way to create a list that allows or denies indexes, with the intent of providing a simple DoS protection. However, their work does not include tests or results. Furthermore, blacklists are very rudimentary forms of DoS protection. Thus, we believe the system proposed in their work does not solve the issue of DoS detection and protection.

4 BACKORDERS

In this chapter, we present our proposed system, BACKORDERS (distriButed deniAl of serviCe attacK detectOr using RanDom forEst in pRogrammable Switches), describing its structure as a whole, as well as detailing each module's characteristics. In Section 4.1, we present an overview of the system, introducing its components. In Section 4.2, we describe the operation of a Random Forest in programmable data planes. In Section 4.3, we further detail BACKORDERS's architecture, stating the process of *training an RF* (Subsection 4.3.1.4), the technique employed for *inserting an RF* in a programmable data plane (Subsection 4.3.2) and the procedure for *extracting features* that are relevant for the classification of a flow (Subsection 4.3.3), as well as describing the remaining modules.

4.1 Approach Overview

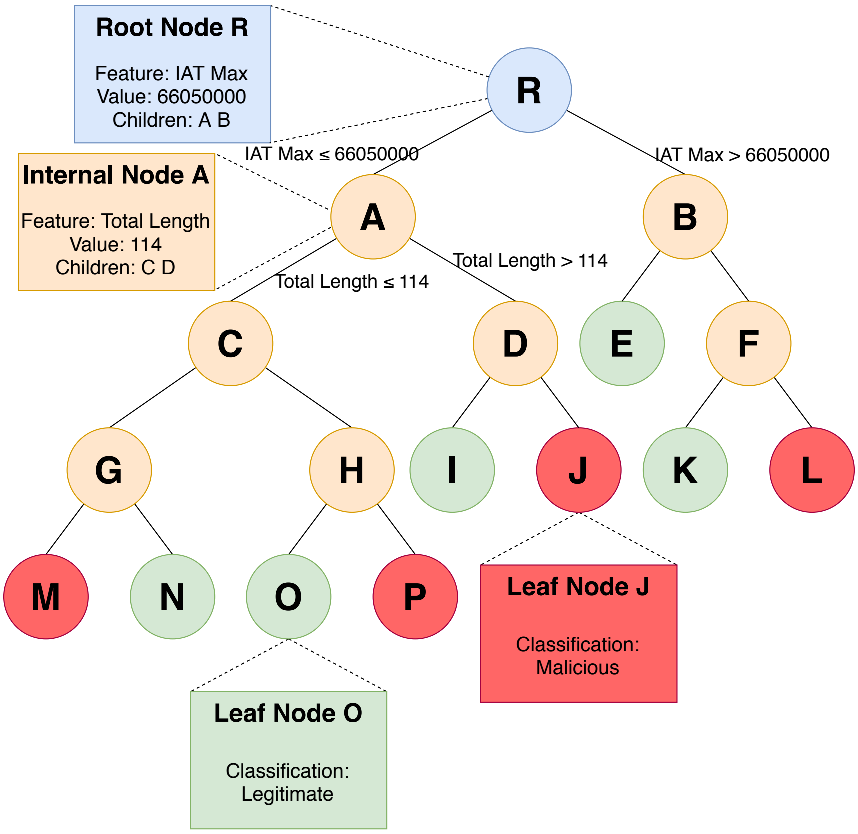
BACKORDERS (distriButed deniAl of serviCe attacK detectOr using RanDom forEst in pRogrammable Switches) is a system for classifying network traffic, with the main purpose of detecting DoS attacks. To achieve this goal, we utilize a Random Forest model to aid in the process of classifying flows. As with any other system that utilizes supervised ML learners, we must have a labeled dataset, with each sample having several features available.

For Random Forests in particular, the training stage has high computational costs, both in terms of CPU and memory utilization. Thus, we elect to perform this step in the control plane, where we can realize the training of an RF on commodity servers, as they tend to have a lot more hardware resources than programmable switches.

Despite the training being computation heavy, the evaluation of a sample is relatively simple. The process of evaluating a sample consists of several comparisons being made, comparing pre-calculated values, called *thresholds*, against values calculated for each sample, called *features*. Although a simple Classification Tree can have up to over a hundred nodes, the number of comparisons done is logarithmic to the number of nodes. Thus, even a tree with a hundred (100) nodes would take no more than seven (7) comparisons, in the worst-case scenario. As such, an RF model can be utilized in P4-enabled switches, performing the classification of network flows at line-rate.

After mapping the RF to a P4-friendly structure, we utilize the ML model for clas-

Figure 4.1: Alternative example of nodes of a Classification Tree



Source: Author

sifying packets of a network flow. We define a flow as a 5-tuple of Source IP, Destination IP, Source Port, Destination Port, and Protocol. As such, for each monitored flow, we calculate several metrics to be utilized as features in the RF. As an RF is an ensemble of multiple Classification Trees, we perform the classification of a flow utilizing each of the trees. Afterwards, we select the class with the highest number of votes, marking the flow appropriately.

4.2 RF in a Programmable Data Plane

Given a Random Forest model, the first step towards performing online classification in the data plane is to map the forest's structure and operations to meet the restrictions imposed by the P4 language, while also appropriately fitting the limited resources of the hardware in programmable switches. In order to achieve these goals, we must take into consideration both the structure of a generic classification tree and the constructs available in the P4 language.

A classification tree is a collection of nodes. Each of these nodes can be either an internal node or a leaf node. To evaluate an **internal node**, it is necessary to first perform a comparison against a threshold value or a series of class values. Depending on the result of the comparison, we must evaluate a different node, repeating this process. As such, internal nodes must know the *feature* to use in the comparison, the *threshold value* to compare it to (or every possible *discrete value*, if that is the case), and it additionally must have a reference to each of its *children*, that is, each possible node that may be subsequently evaluated. Figure 4.1 shows a visual example of an internal node 'A', where we can observe its *feature* (Total Length), *threshold* (114), and *children* ('C' and 'D'). For the other kind of node, we may have a **leaf node**. Leaf nodes are a bit simpler, as all the information they must contain is a *classification* for samples whose path leads to that *leaf*, as we can see for the leaf nodes 'J' and 'O'.

Common implementations of Classification Trees in commodity servers utilize *recursion*. This is done by having one instance of a class Node, while executing its own evaluation function, invoke one of its children's evaluation function. In order to do this, a function must have a reference to itself in its code, being classified as a recursive function. Algorithm 1 shows a pseudo-code with recursion, where the function *Eval* has a reference to itself. Although recursive functions are often considered intuitive for certain scenarios, they remove the ability of a compiler to know ahead of time how many times a function is going to be invoked. As such, recursion is not currently supported by the P4 language, given that programmable switches must meet strict time constraints to operate at line-rate.

Algorithm 1: Example of a recursive function

`Eval`($Node$, $Sample$)
if *Node is leaf* **then**
    return *Classification*
else if *Sample's relevant feature* $\leq$ *Node's threshold* **then**
    `Eval`($Child1$, $Sample$)
else
    `Eval`($Child2$, $Sample$)

In order to map a Classification Tree to a programmable data plane, we must find a way to represent and sequence the evaluation of nodes successively, without using recursion. Additionally, nodes may not contain direct references to a structure of another node. Thus, we propose a way to map the information that each node must hold into entries of a match+action table, fitting in the data plane.

Given a classification tree node, such as the one in Figure 4.1, we map its relevant *feature*, *threshold*, and *children* to a match+action table. In our approach, each node has

Table 4.1: Example of a partial Classification Tree mapped to match+action table

Match Value	**Action**	**Parameters**		
Node Identifier		**Threshold**	**Child 1**	**Child 2**
R	compare_iat_max	66050000	A	B
A	compare_total_length	114	C	D
B	compare_feature_B	y	E	F
H	compare_feature_H	z	O	P

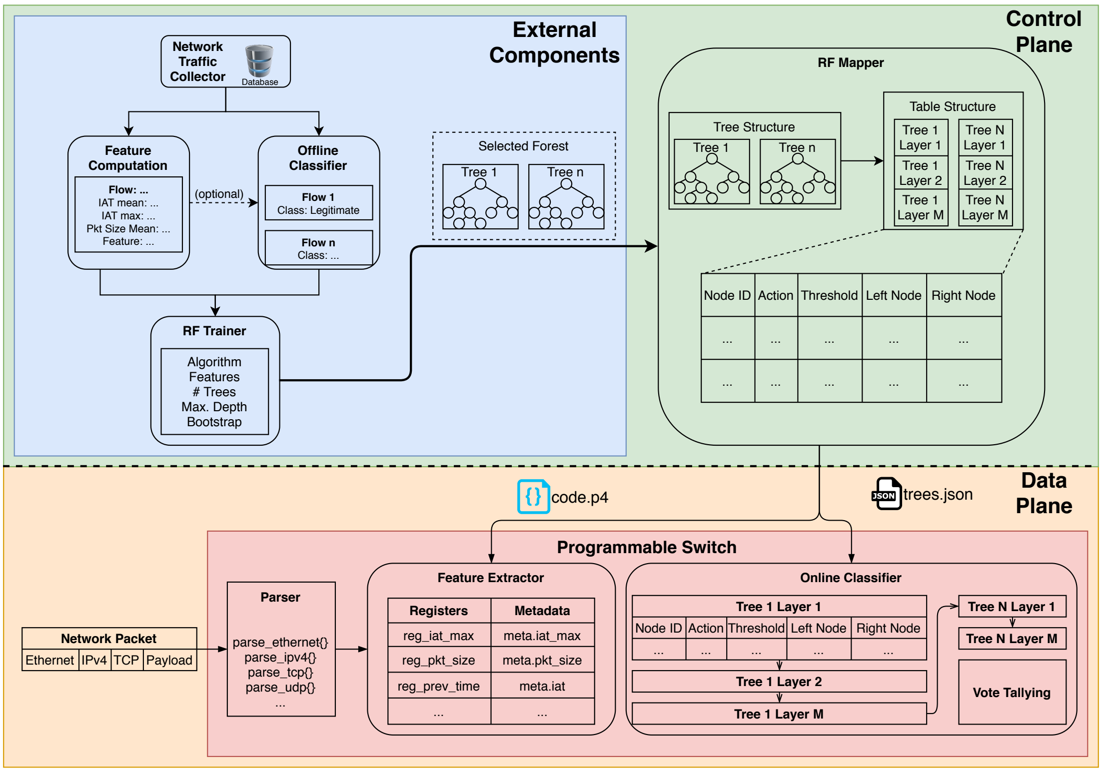
Source: Author

a unique identifier, such as 'A' and 'B'. Still using the example of the aforementioned figure, we can observe that the root node has an identifier 'R'. A visual representation of the match+action table entry that our *node R* would be mapped to can be seen in Table 4.1. As such, this mapping would match its *identifier* 'R', invoking the action *compare_iat_max*, passing as parameters the *threshold* 66050000 and its children, where the *first child* is 'A' and the *second child* is 'B'. Once invoked, the action would compare the appropriate feature to the threshold passed as parameter, indicating that the next node to be evaluated is 'A' if *iat_max* has a value less than or equal to 66050000, or the node 'B' if the value is greater than 66050000.

4.3 BACKORDERS Architecture

BACKORDERS is divided into a series of modules, each one with a separate purpose. Figure 4.2 shows how these modules are organized. Our system depends on a series of external components, highlighted in the figure, which will be briefly explained in Subsection 4.3.1. Aside from the external components, BACKORDERS has a module that runs on the control plane, the *RF Mapper* module, which will be detailed in Subsection 4.3.2. In order to realize the classification of network traffic, BACKORDERS implements two modules in the data plane, (1) a *feature extractor* module, which will be thoroughly explained in Subsection 4.3.3, and (2) an *online classifier* module, responsible for orchestrating the multiple Classification Trees implemented in the data plane, which will be detailed in Subsection 4.3.4.

Figure 4.2: Architecture of BACKORDERS



Source: Author

4.3.1 External Components

As mentioned in Section 4.3, for the correct operation of our proposed system we must have a series of external components, all of which execute on the control plane. These components include a *network traffic collector* module, a module responsible for *feature computation*, an *offline classifier* module, and an *RF trainer* module. However, these modules are not the main contribution of this work, as they have been studied by many different researchers in the past. Despite this, as these modules are still relevant to BACKORDERS, we will briefly explain their purpose and general operation in this subsection.

4.3.1.1 Network Traffic Collector

The *network traffic collector* module is responsible for collecting network traffic to be used by the other modules. This is a necessary first step towards training classifier models, such as a Random Forest. For use in ML learners, the network traffic collected (often referred to as dataset) must be sizeable and represent the different types of traffic that are present in the network. For this task, network operators may decide to collect traffic from their organization, utilizing tools such as tcpdump or any other tool that allows monitoring and storing network packets. In many scenarios, it is important to realize the anonymization of the data collected. However, performing this step while maintaining some of the important characteristics of the original data may be challenging.

4.3.1.2 Feature Computation

After enough network traffic data is collected, it is sent to the module responsible for *feature computation*. There, each collected packet is aggregated into the respective network flow that it is a part of. Afterwards, relevant features must be computed, so that they may be utilized by the next module. The set of features to be computed and the methods for doing so depend on the necessities of the ML learner, along with the capabilities of the programmable switch. For instance, consider a feature that calculates the Fourier transform of a particular statistical value (SANTOS DA SILVA et al., 2015). While this feature may contribute to the classification of network traffic, it would be very complex and difficult to implement in a programmable data plane, potentially having too high of a computational cost to justify the gain in classification accuracy. As such, it is important to keep in mind the limitations of P4-enabled switches. An example of a tool for the extraction of features is CICFlowMeter (LASHKARI et al., 2020), offering dozens of flow statistics.

4.3.1.3 Offline Classifier

Once enough network traffic has been collected, it must be classified before being used to train an ML model. The *offline classifier* module, as opposed to the *online classifier* module in the programmable switch, can perform its work in parallel with the *feature computation* module, if the classifier module does not require these features to realize its task. The *offline classifier* module has an equally important and difficult task, as the classifications done here will be used by the ML learner as ground-truth, ultimately affecting its accuracy when classifying traffic in practice, after its training and evaluation. The approaches for realizing offline traffic classification include DPI, neural networks, and many other techniques. However, considering the difficulty and importance of this task, there is not a de facto approach for classifying unknown traffic with extremely high accuracy, even when done in an offline manner, where higher computational costs and longer execution times are tolerated. As such, an approach utilized by some researchers is to, rather than attempt to classify unknown traffic, generate their own traffic. An example of a dataset artificially generated is the CIC-DDoS2019 dataset (SHARAFALDIN et al., 2019).

4.3.1.4 RF Trainer

Following the acquisition of an appropriate dataset, we utilize it to train an RF model in the *RF trainer* module. Along with the labeled network traffic and extracted features, random forests have a number of meta-parameters, such as the number of classifiers (trees) and the number of features per tree. Additionally, there are meta-parameters for the trees, such as maximum depth and the algorithm for selecting the appropriate number of features for each tree. These meta-parameters aim to introduce diversity in the multiple Classification Trees that compose the trained RF, as explained in Subsection 2.3.2. The algorithm implemented in this module for performing the induction of Classification Trees aims to automatically select the most relevant features first, as explained for the C4.5 algorithm described in Subsection 2.3.1. While in Subsection 2.3.1 we presented only the C4.5 algorithm, which utilizes normalized information gain, there are different approaches for classification tree induction, such as the CART algorithm (BREIMAN et al., 1983). Although C4.5 is a very popular algorithm, there are situations where a different algorithm may produce an RF better suited for certain datasets. However, the general data structure of classification trees is similar regardless of the algorithm utilized for induction. Thus, the algorithms are interchangeable, as long as the *RF mapper* module receives the appropriate input, as it is the next module in the workflow.

4.3.2 RF Mapper

The first main module of BACKORDERS is the *RF mapper module*, which runs on the control plane. This module, as briefly explained in Section 4.2, is responsible for taking an input describing the structure of a Classification Tree and to map it into an output match+action table to be inserted in the P4-enabled switch. As we are working with a Random Forest, which contains multiple CTs, we repeat the process for each tree in the forest, generating multiple match+action tables.

Recapitulating the explanations in Section 4.2, the information contained in an **internal node** is its *identifier*, *feature* to be used, *threshold*, and *children*. For a **leaf node**, however, the only information we need is its *identifier* and *classification*, as these types of nodes do not have children or perform any comparisons.

For *identifiers*, our approach utilizes **numeric identifiers**, as opposed to what was shown in Section 4.2 for easier understanding. Each node in a tree has a unique identifier, starting with 0 for the root, with increments of 1 for the subsequent nodes. As such, a tree with n nodes will have identifiers ranging from 0 to $n-1$. While each node must have a unique identifier within the tree it belongs to, nodes of different trees can have repeated identifiers. Thus, even as the root node of a CT i will have the identifier 0, the root node of another CT j will also have the identifier 0.

Internal nodes have their *feature* field mapped into a specific P4 action. Ergo, every node that utilizes a specific feature will invoke the same action. As an example, as shown in Table 4.1, a node that utilizes the number of packets of that flow, a feature named 'pkt_num', will invoke the corresponding 'compare_pkt_num' action, which will realize the comparison of the 'pkt_num' feature of that flow against the *threshold* parameter in the match+action table entry. This will be true not only of node 'R', in the aforementioned example, but also of every other node that utilizes 'pkt_num' as its feature.

Still on the topic of the information contained in **internal nodes**, our system only implements numeric features at this time. While at first it may seem restrictive, we believe that for the specific task of network traffic classification it does not affect the quality of the learner model. This is due to the fact that fields of network protocol headers are generally interpreted as either a number or a *boolean* flag, that is, a variable that only assumes values False (0) or True (1). Thus, as the system currently only utilizes numeric attributes, mapping a node's *threshold* into the data plane is trivial, simply inserting it as a parameter of the corresponding entry of the match+action table.

The last information that the *RF mapper* module must map from an internal node into the programmable data plane is its children. As previously explained in Section 4.2, P4 does not allow recursion. As such, we break the naturally recursive structure into entries of a table. In order to do this, we utilize the same match+action table entry where the other information is, passing as parameters to the compare feature action (such as 'compare_pkt_num') the unique identifier of this node's children. Taking into account our proposed system's current characteristic of strictly utilizing numeric *threshold* values, we know that each internal node will always have two children. Thus, we pass two parameters to the aforementioned action, the identifiers of the 'left' child and of the 'right' child nodes.

Table 4.2: Example of leaf nodes mapped to match+action table

Match Value **Node Identifier**	**Action**	**Parameters** **Classification**
25	classify_flow	MALICIOUS
26	classify_flow	LEGITIMATE
32	classify_flow	LEGITIMATE
46	classify_flow	MALICIOUS

Source: Author

Now that the mapping of an **internal node** has been explained, the only information missing is the one contained in a **leaf node**. As previously explained, this information only includes a unique *identifier* and a *classification*. The process of mapping a **leaf node's** *identifier* is analogous to the mapping of an **internal node's** *identifier*, as was previously explained. The other information we must map is its *classification*. For this, we use a slightly different format of a match+action table entry. Table 4.2 shows an example of match+action table entries containing the information of leaf nodes, where we can still observe an *identifier*, however now we always utilize the same action, 'classify_flow', passing as parameter the *classification* of this leaf node. While the table shows a *classification* being either 'MALICIOUS' or 'LEGITIMATE', in order to optimize the system for higher performance in a programmable switch, we utilize the values 0 (legitimate) and 1 (malicious).

4.3.3 Feature Extractor

Unlike the previous modules that were explained, the *feature extractor* module is located in the programmable data plane of a P4-enabled switch. This module realizes computations for each incoming packet, with its first task being the identification of the flow the arriving packet belongs to. Afterwards, it utilizes **metadata** information, such as the *ingress time* of the packet and its *packet length*, along with any relevant **header** fields, such as *urgent* and *push* flags of an IPv4 header, or *window size* value of a TCP header, in order to extract features to be utilized by the RF. In Subsection 4.3.3.1 we enumerate the supported features and describe the computation of the simpler ones. In Subsection 4.3.3.2 we detail the technique employed to approximate moving means, as utilized in this work.

Table 4.3: Features implemented

<table>
<tr><td colspan="2">Destination Port</td></tr>
<tr><td colspan="2">Flow Duration</td></tr>
<tr><td colspan="2">Packet Count</td></tr>
<tr><td colspan="2">Header Length Sum</td></tr>
<tr><td colspan="2">Initial Window</td></tr>
<tr><td colspan="2">ACT Data Count</td></tr>
<tr><td>PSH</td><td rowspan="2">Flag Count</td></tr>
<tr><td>URG</td></tr>
<tr><td rowspan="4">Packet Length</td><td>Total</td></tr>
<tr><td>Minimum</td></tr>
<tr><td>Maximum</td></tr>
<tr><td>Mean</td></tr>
<tr><td rowspan="4">Inter-Arrival-Time</td><td>Total</td></tr>
<tr><td>Minimum</td></tr>
<tr><td>Maximum</td></tr>
<tr><td>Mean</td></tr>
<tr><td rowspan="4">Segment Size</td><td>Total</td></tr>
<tr><td>Minimum</td></tr>
<tr><td>Maximum</td></tr>
<tr><td>Mean</td></tr>
</table>

Source: Author

4.3.3.1 Simple Features

Our system currently supports twenty features, as shown in Table 4.3. As we can observe, some features are based on values from the **IPv4 header**, such as *PSH flags*. Other features utilize values from the **TCP** or **UDP headers**, such as *destination port*. Additionally, there are features that utilize **metadata**, such as the ingress timestamp of each packet, to calculate values such as *inter-arrival-time* of packets of a flow. In this subsection, we will explain every feature in that table, as well as detail the computation process of all of them except for means, which will be discussed in Subsection 4.3.3.2.

- *Destination port* is extracted from the TCP or UDP header (whichever is present) of the current packet. This value is also stored in a register, as it is utilized to separate different flows;
- *Flow duration* is the time elapsed between the arrival of the first packet of that flow and the current time. To store the timestamp of the first packet of each flow we utilize a register. To approximate the current time we utilize the standard metadata ingress timestamp of the current packet;
- *Packet count* is the number of packets of that flow observed by the switch. This

value is stored in a register and is incremented by one for each arriving packet belonging to that flow;

- *Header length sum* is the sum of the length of headers of each packet of that flow, in bytes. This value is stored in a register and is incremented by the length of the headers of each arriving packet belonging to that flow;
- *Initial window* is extracted from the TCP header of the first packet of a flow if the header is present. This value is stored in a register and is not changed after the first packet of that flow;
- *ACT Data Count* is the number of packets of that flow that had at least one byte of payload. This value is stored in a register and is incremented by one for each arriving packet belonging to that flow that has a payload length of at least one byte;
- *PSH Flag Count* is the number of packets of that flow that had the PSH flag set, extracted from the IPv4 header. This value is stored in a register and is incremented by one for each arriving packet belonging to that flow that has the PSH flag of the IPv4 header set (not zero);
- *URG Flag Count* is the number of packets of that flow that had the URG flag set, extracted from the IPv4 header. This value is stored in a register and is incremented by one for each arriving packet belonging to that flow that has the URG flag of the IPv4 header set (not zero);
- *Packet length* is calculated based on the number of bytes of each packet of that flow:
  - The *total packet length* of a flow is the sum of the length of every packet of that flow. This value is stored in a separate register. It is incremented by the length in bytes of each arriving packet belonging to that flow;
  - The *maximum packet length* of a flow is the highest value of the length of every packet of that flow. This value is stored in a separate register. It is updated whenever the packet length of an arriving packet belonging to that flow is greater than the previously stored value;
  - The *minimum packet length* of a flow is the smallest value of the length of every packet of that flow. This value is stored in a separate register. It is updated whenever the packet length of an arriving packet belonging to that flow is lesser than the previously stored value;
  - The *mean packet length* of a flow is the moving average of every packet of that flow. This value is stored in a separate register. It is updated for each arriving

packet belonging to that flow. The exact method for approximating this value is discussed in Subsection 4.3.3.2;

- *Inter-arrival-time* (IAT) is the time elapsed between the arrival of the previous packet of that flow and the next, at the current time. To store the timestamp of the previous packet of each flow we utilize a register, updated after the calculation of the IAT. To approximate the current time we utilize the standard metadata ingress timestamp of the current packet. Each of the four different features that refer to IAT, *total IAT*, *maximum IAT*, *minimum IAT* and *mean IAT* are analogous to the previously explained features about packet length;
- *Segment size* is the length of the payload of each packet of that flow. To calculate this value we utilize the standard metadata packet length of the packet, along with pre-calculated values of lengths of different headers. Each of the four different features that refer to segment size, *total*, *maximum*, *minimum* and *mean segment size* are analogous to the previously explained features about packet length.

4.3.3.2 Approximating Means

As explained in Section 2.2, the P4 language does not support divisions. Thus, calculating the average values is not a trivial task, as it requires the computation of the division of a sum by its number of elements. Due to this restriction, many researchers utilize Exponentially Weighted Moving Averages (EWMA), by setting a weight of $\frac{1}{n}$, with n being a power of two. This is done because, in computer systems based on bits, divisions by a divisor that is a power of two can be simulated by performing bit-shifts, that is, shifting the value of the internal representation used by computers of that number. However, EWMA is not the same as the moving averages normally calculated, and as such it produces different results from what is used for training the classifiers of the RF.

In order to minimize the difference between the computed approximation and the real moving average, we propose a new technique to be employed in lieu of EWMA. Our algorithm requires the computation of the *total sum of values* and the *number of values*, along with storing the previously computed *approximated mean* and an additional auxiliary value that we refer to as "*approximate sum*". While it may initially seem like a big overhead compared to utilizing a technique such as EWMA, our system already computes and stores the total sum of values and the number of values, as they are utilized as features for our RF. Thus, the only additional memory cost our algorithm introduces to our system is an extra stored value, the *approximate sum*.

Given i values V_i, $S_e(i)$ is the exact sum of every value V_i (Equation 4.1). To calculate an approximate mean M_a, we also need to calculate an approximate sum, S_a. Each of these values is calculated at each step i, that is, as each new sample value V is introduced. Both of these values, M_a and S_a, have conditional formulas, having their values calculated based on $S_e(i)$ at certain i. First, we can see, in Equation 4.2, how to calculate the approximate sum S_a for a given i. When i is a power of two, such as 1, 2, 4, and so on, $S_a(i)$ is simply the exact sum, $S_e(i)$, which is trivial to calculate (Subequation 4.2a). In every other case, we utilize the previous approximate sum, $S_a(i-1)$, the previous approximate mean, $M_a(i-1)$ and the new sampled value, V to calculate the new approximate sum (Subequation 4.2b).

$$S_e(i) = \sum_{i=1}^{N} V_i \tag{4.1}$$

$$S_a(i) = \begin{cases} S_e(i), & \exists n \in \mathbb{N}, 2^n = i \quad \text{(4.2a)} \\ S_a(i-1) - M_a(i-1) + V, & \nexists n \in \mathbb{N}, 2^n = i \quad \text{(4.2b)} \end{cases}$$

Equation 4.4 shows the process for calculating the approximate mean $M_a(i)$ at a given step i. When i is a power of two (Subequation 4.4a), we simply calculate the "exact" mean, by dividing the exact sum, $S_e(i)$ by i. Despite divisions not being available, we can obtain a similar result by shifting the dividend by an appropriate number of bits. As such, our goal is to always force divisions by a power of two. However, when i is not a power of two, we utilize an approximate sum $S_a(i)$, as shown in Equation 4.2. Additionally, for approximating means when i is not a power of two, we calculate a value referred to as $prev\_pow2(i)$. This value is calculated as shown in Equation 4.3, and it is the highest power of two that is lesser than i.

$$prev\_pow2(i) = max\ 2^n < i, n \in \mathbb{N} \tag{4.3}$$

Considering the values calculated previously, the exact sum of i values, $S_e(i)$ (Equation 4.1), the approximate sum of i values, $S_a(i)$ (Equation 4.2) and the highest power of two that is smaller than i, $prev\_pow2(i)$ (Equation 4.3), we can finally approximate means for i that is not a power of two. This is done by dividing the approximate sum, $S_a(i)$ by $prev\_pow2(i)$, which can be done with bit shifts, as $prev\_pow2(i)$ is a power

Table 4.4: Example of the values during an execution of the approximation algorithm

i	V_i	$S_e(i)$	$S_a(i)$	$M_a(i)$	**Mean**	**Formula:** $S_a(i)$	**Formula:** $M_a(i)$
8	15	160	160	20	20	$S_e(8)$	$S_e(8)/8$
9	25	185	165	20.625	20.5	$S_a(8) - M_a(8) + V_9$	$S_a(9)/prev\_pow2(9)$
10	10	195	154.375	19.29875	19.5	$S_a(9) - M_a(9) + V_{10}$	$S_a(10)/prev\_pow2(10)$

Source: Author

of two, as shown in Subequation 4.4b. The idea behind utilizing an approximate sum, $S_a(i)$, instead of the exact sum, $S_e(i)$, is that the exact sum has i values added, whereas we calculate the approximate sum in such a way that it has a value closer to the sum of $prev\_pow2(i)$ values.

$$M_a(i) = \begin{cases} \dfrac{S_e(i)}{i}, & \exists n \in \mathbb{N}, 2^n = i \quad (4.4a) \\ \dfrac{S_a(i)}{prev\_pow2(i)}, & \nexists n \in \mathbb{N}, 2^n = i \quad (4.4b) \end{cases}$$

For better understanding, we will provide an example, as shown in Table 4.4, with step-by-step computation. As 1, 2, and 4 are powers of two, starting with the first value would simply result in the usual calculation of moving means for three out of the four steps. Thus, we shall start with the computation of an example with the calculation of the eighth value. First, assume that $V_8 = 15$ and $S_e(7) = 145$. As $i = 8$ is a power of two, the approximate sum $S_a(i)$ will be the same as the exact sum $S_e(i)$ and the approximate mean $M_a(i)$ will be the simple division of the exact sum by i. Thus, $S_e(i) = S_a(i) = 145 + 15$, which results in 160. Consequently, $M_a(i) = \frac{160}{8}$, calculated as in Subequation 4.4a, resulting in 20. This is the way to calculate exact means when i is a power of two.

However, when a new value $V_9 = 25$ arrives, with $i = 9$, we must calculate differently. Starting with Equation 4.3, $prev\_pow2(9) = 8$, as $2^3 = 8$, and $8 < 9$. Next, we approximate the sum for $i = 9$, following Subequation 4.2b, we arrive at

$$S_a(9) = S_a(8) - M_a(8) + V_9 \tag{4.5}$$

replacing the appropriate values, $S_a(8)$, $M_a(8)$ and V_9

$$S_a(9) = 160 - 20 + 25 = 165 \tag{4.6}$$

and thus, following Subequation 4.4b, we obtain

$$M_a(9) = \frac{S_a(i)}{prev\_pow2(i)} \tag{4.7}$$

where, by replacing the appropriate values, as calculated previously, we can calculate the approximate mean as

$$M_a(9) = \frac{165}{8} = 20.625 \tag{4.8}$$

As we can see, 20.625 is relatively close to the exact mean, as shown in Table 4.4, equating to 20.5. We will briefly show the next step, with $i = 10$ and $V_{10} = 10$, to highlight that, by having an approximate sum calculated by removing the previous approximate mean, our approach does a commendable job of approximating the real mean. As the formulas were already shown and explained for $i = 9$, we will omit the step where we replace the original equation with the appropriate i, and will instead only show the corresponding values on the right side of each equation. First, $prev\_pow2(10) = 8$. Next, we calculate the approximate sum, as

$$S_a(10) = 165 - 20.625 + 10 = 154.375 \tag{4.9}$$

finally, we obtain the approximate mean as

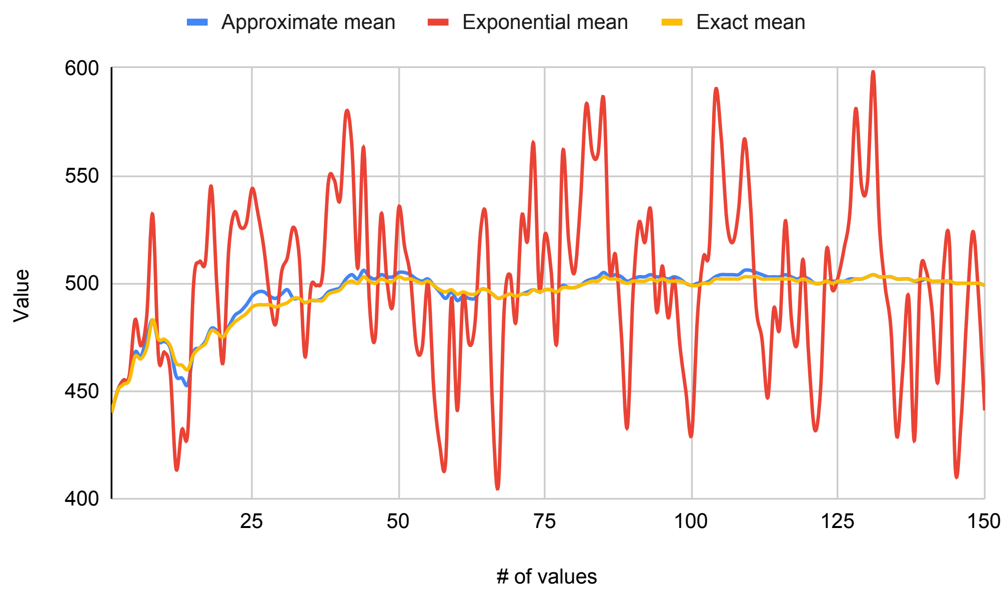
$$M_a(10) = \frac{154.375}{8} = 19.296875 \tag{4.10}$$

which is relatively close to the exact mean, 19.5.

To further evaluate the effectiveness of our approach, we realized a comparison between our algorithm for approximating means and EWMA, which is sometimes used in P4-enabled systems as an alternative. For these tests, we utilized a tool (HAAHR, 2020) to generate over a hundred random numbers following a Gaussian distribution with mean 500 and standard deviation 65.

First, in Figure 4.3, we plot the calculated value of our approach (in blue), EWMA (in red) and the exact value (in yellow). As we can observe, our approach (in blue), better approximates the exact value (in yellow). An often undesirable characteristic of EWMA is its sensitivity to new values that happen to be outliers. This issue is amplified when using a weight of 0.5, as used in (BUSSE-GRAWITZ et al., 2019). To compare that behavior with our approach, we also utilized the weight of 0.5 for EWMA in our tests. We can observe this behavior as the EWMA value (in red) varies abruptly, increasing its

Figure 4.3: Mean values computed by different approaches



Source: Author

value significantly higher than the exact value, observed in the curve in yellow.

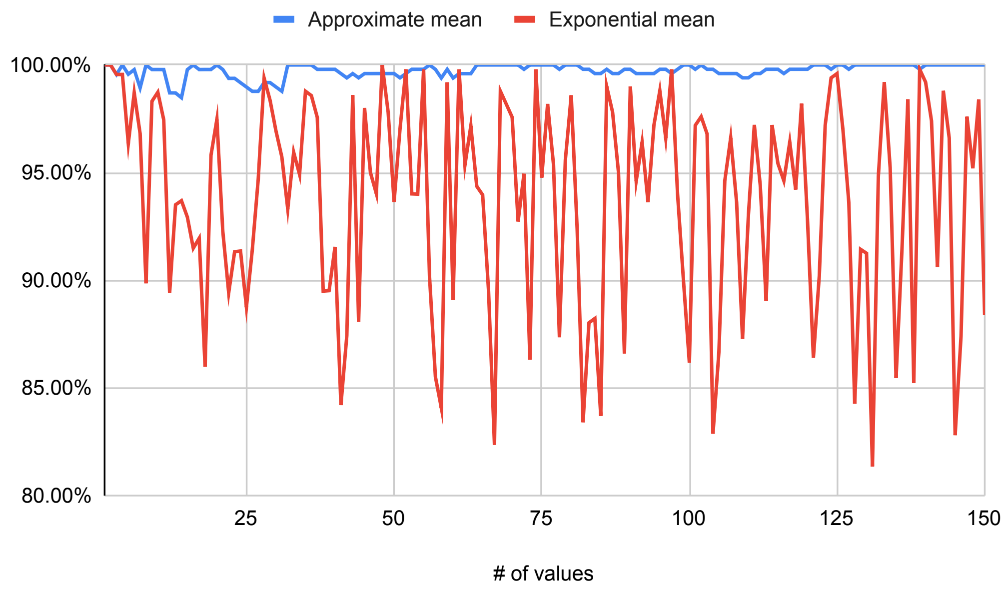
In Figure 4.4, we plot the accuracy obtained by our approach (in blue) and EWMA (in red), in relation to the exact value. This plot better shows that our approach (in blue) has significantly higher accuracy than EWMA (in red). Additionally, we can see very high dips in accuracy which, as previously explained, is likely due to the high sensitivity to outliers that EWMA suffers from when utilizing a weight such as 0.5.

Lastly, Figure 4.5 shows the average and minimum accuracy obtained by our approach (in blue) and EWMA (in red). As expected from the behavior of EWMA previously explained, its minimum accuracy (in red) is significantly lower than our approach's (in blue). This is due to the high sensitivity to outliers when using a high value for its weight. Additionally, we observe that even the lowest accuracy obtained by our approach (in blue) is quite high. Thus, we conclude that our approach does a good job of approximating the real mean.

4.3.4 Online Classifier

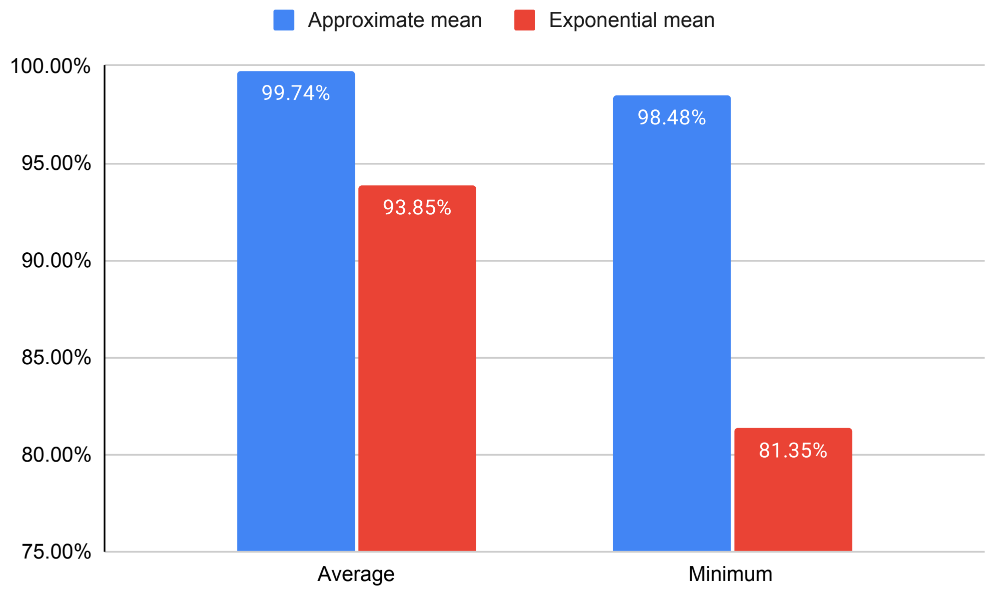
The last module of our system is the *online classifier*. This module is responsible for orchestrating the multiple Classification Trees that compose our Random Forest in order to perform the classification of a network flow. The *online classifier* is located

Figure 4.4: Comparison between our approach's and EWMA's obtained accuracy



Source: Author

Figure 4.5: Comparison between our approach's and EWMA's average and minimum accuracy



Source: Author

in the ingress processing block of the P4-enabled switch, where the logic for utilizing multiple classification trees is done.

Figure 4.6 shows a representation of the control flow of our system, BACKO-RDERS, in the programmable data plane. For the approximation of means, as previously explained in Subsection 4.3.3, we must calculate the power of two that is smaller than i, when i is not a power of two. To efficiently find the value of $prev\_pow2(i)$, we utilize a match+action table, populated with values of powers of two, along with masks to appropriately match the value i. However, some of our features calculate features over $i - 1$ values, rather than over i values. An example of a feature that utilizes $i - 1$ values is the inter-arrival-time, including maximum, minimum, total and mean. This is due to the fact that the IAT is the time between the arrival of two consecutive packets. As such, there is no IAT for a flow with a single packet.

Thus, before the *online classifier* requests the extraction of features, we must calculate $prev\_pow(i)$ utilizing the appropriate table. However, as the features calculated around IAT utilize $i - 1$ values, we first calculate $prev\_pow(i - 1)$. This calculated value is then stored in a user-defined global metadata variable (as opposed to standard metadata), so that it can be used by the *feature extractor* to approximate the mean IAT. Next, this process is repeated with i, calculating $prev\_pow(i)$, to be used by the *feature extractor* to approximate the remaining mean values, such as *mean packet length* and *mean segment size*.

Once the *feature extractor* performs its function, after receiving a request from the *online classifier*, we are ready to perform the classification of a network flow utilizing our RF. The *RF mapper* module maps the trained RF into several tables per tree, separating nodes by their depth. This is done because P4 does not allow the control block to invoke the same table multiple times. Thus, by separating nodes by their depth, we ensure that invoking the same table multiple times will never be necessary.

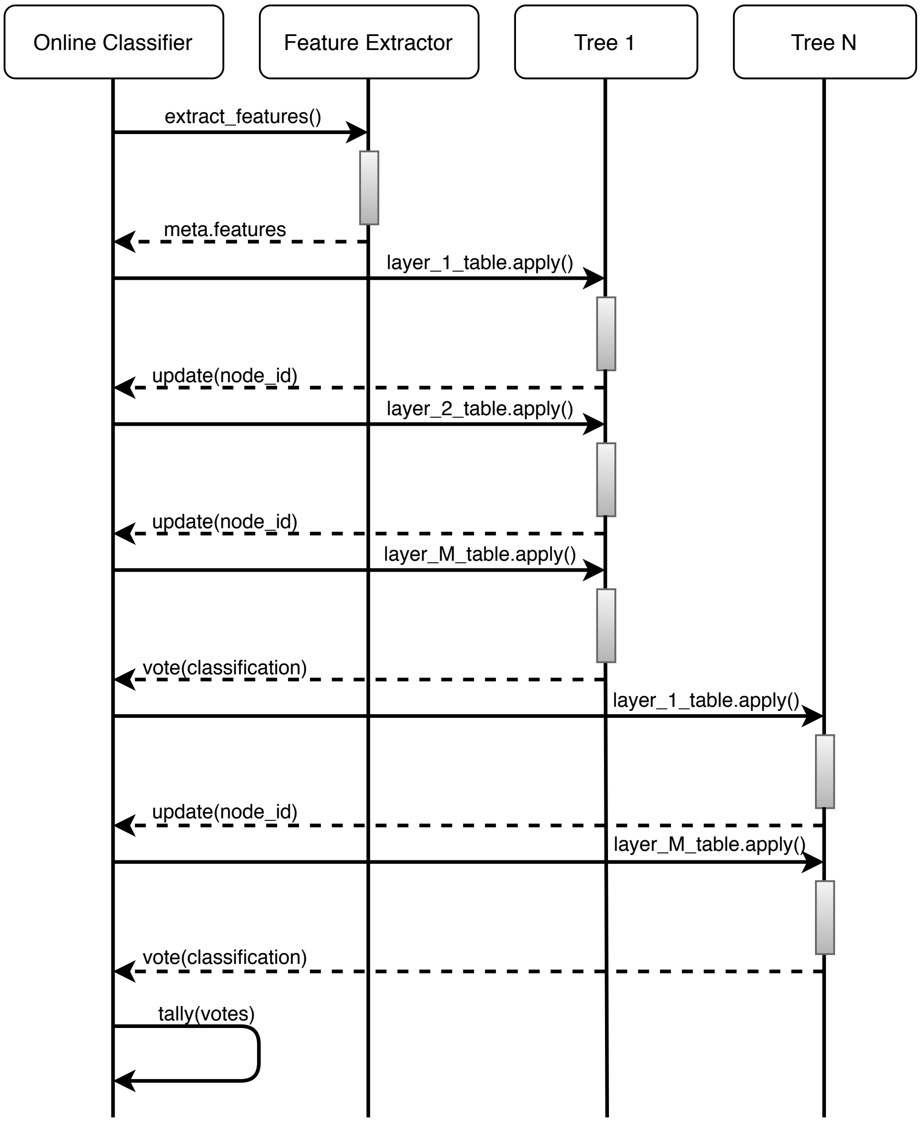
Revisiting Figure 4.1 as an example, we know that during the classification of a sample, nodes 'A' and 'B' will <u>never both</u> be evaluated. This is due to the fact that after the evaluation of node 'R', only <u>one</u> of either path will be taken. Analogously, for nodes 'C', 'D', 'E' and 'F', only <u>one</u> of them will be evaluated. Thus, we can easily split a single tree into multiple tables, separating nodes by layer (or depth), having every node of a given layer L be in the same table.

As such, the *online classifier* must invoke each of the tables that compose each Classification Tree in our RF. First, the *online classifier* sets a user-defined global metadata variable, $node\_id$ to indicate that our first node has the *identifier* 0, as every root node has that specific identifier. Afterwards, we apply the table of the first tree with depth 1, where the metadata variable with value 0 will be matched with the *identifier* of the root, also 0. After the values are matched in the corresponding match+action table entry, the appropriate action $compare\_feature\_X$ will be called, comparing feature 'X' to the *threshold* present in the match+action table entry. This action, after realizing the appropriate comparison, will set the same metadata variable, $node\_id$, to the appropriate value. Once again referencing Figure 4.1 as an example, after the metadata $node\_id$ matches with the *identifier* of root node 'R', the action $compare\_iat\_max$ will be invoked, passing as parameters the *threshold* 66050000 and children 'A' and 'B'. Thus, if the iat_max feature of this specific flow is smaller or equal to 66050000, the action will set the next value of $node\_id$ as the (numeric) *identifier* of node 'A'. Otherwise, it will set $node\_id$ to the (numeric) *identifier* of node 'B'.

The process mentioned above is repeated for each layer of the first tree, invoking as many tables as the maximum depth of the tree. However, as not every *leaf node* is in the last layer of the tree, we utilize a user-defined global metadata variable, $is\_classified$, as a flag. This flag is checked before applying subsequent tables, avoiding unnecessary applications of tables. Once a *leaf node* has its *identifier* matched against the global metadata variable $node\_id$, the appropriate $classify\_flow$ action is invoked, passing as a parameter the *classification* of that node. In this action, along with setting the value of the $classification$ user-defined global metadata variable, we set the previously mentioned $is\_classified$ flag as true, avoiding unnecessary processing for this tree.

The process of matching $node\_id$ against each table containing one layer of the tree is done until a classification is obtained from that particular tree. Afterwards, we reset $node\_id$ to 0, and start the classification of the same sample in the next tree, repeating the entire process explained above. This is done until every tree has provided a classification for the sample. Once that has happened, we count the votes of each tree, stored in local variables after the classification of each tree is obtained. Lastly, we select the classification with the highest number of votes as the final classification of that flow.

Figure 4.6: Online Classifier Control Diagram



Source: Author

5 IMPLEMENTATION AND EVALUATION

In this chapter, we present the implementation and evaluation of BACKORDERS. In Section 5.1 we describe the characteristics of the prototype that was implemented. In Section 5.2 we present the methodology employed in the experiments, specifying the dataset utilized and the meta-parameters employed in each RF. In Section 5.3 we present the results obtained, comparing different configurations. In Section 5.4 we discuss some aspects of the real-world applicability of our system.

5.1 Prototype

To evaluate our proposed system, we implemented an initial prototype. The prototype includes modules that run on the control plane and other modules that run on the data plane. In the control plane, we utilized an available ML library in Python 3 for the external module *RF trainer*, along with our own implementation of the *RF mapper*, also in Python 3. In the data plane, we utilized the P4 language in order to implement the *feature extractor* and *online classifier* modules.

In the control plane, we utilized the *sklearn* Python 3 library to train multiple Random Forests in the *RF trainer* module. This library provides a high degree of customization of ML learners through several parameters that can be adjusted. Additionally, *sklearn* provides Python library modules to extract diverse classification metrics, which were utilized to compare the multiple trained learners. Once the training was complete, we exported the classifiers in text format, utilizing a method also provided by the library. Afterwards, we implemented the logic of the *RF mapper*, as described in Section 4.3.2, in Python 3. This module generates JSON configuration files for the insertion of trees and parts of P4 code for the definition of the multiple tables that contain the trees.

In the data plane we implemented a prototype of the *feature extractor* and the *online classifier* modules, as described in Section 4.3.3 and Section 4.3.4, respectively. The implementation was done according to the P4-16 specification, with BMv2 as the target model, specifically the *simple_switch_grpc* variation of BMv2.

In order to calculate every feature listed in Table 4.3, we utilized 23 registers for values, along with 5 registers to store the 5-tuple that defines a flow, Source and Destination IP Addresses, Source and Destination Ports and Protocol. For the feature registers, not including the registers to store the 5-tuple, we utilized 48 bits for features related to

IAT, as timestamps in P4-16 utilize 48 bits. For the remaining features, we utilized a common value of 32 bits, in order to facilitate implementation and avoid possible overflow issues. Table 5.1 shows a list of values and their size in bits, while also explaining if the value is stored in a register and if it is copied into a metadata variable. For instance, the destination port is currently stored in a register to identify the flow. However, its value is not copied into a metadata variable, as this information is in the TCP or UDP header. The initial window, however, is stored in a register, as this value is only present in the first packet. Additionally, it is copied into a metadata variable to be used by the classification trees. The length of the current packet is already present in the standard metadata made available by BMv2, thus it is not copied or stored. This brings our total memory usage to 848 bits (excluding the destination port), or 106 bytes, per-flow. This number implies very high memory usage by our system. However, we discuss possible optimizations in Section 6.2. In this initial prototype, we aimed to provide an implementation for the calculation of every feature, regardless if it was utilized by the inserted forest or not. A potential optimization will be discussed in Section 6.2.

The *online classifier* module invokes the tables that compose each tree, repeating the process for every tree. As trees are coded into match+action tables, we utilize metadata variables with the values of previously calculated features (by the *feature extractor* module), so the actions that realize the comparison of a feature against a threshold can easily access the feature's value. The number of bits utilized by metadata variables is similar to the number of bits utilized by registers, however, the metadata variables are not stored per-flow. Additionally, the *online classifier* invokes two tables, one to calculate $prev\_pow2(i-1)$ and one to calculate $prev\_pow2(i)$. This is necessary as the features related to IAT utilize $i-1$ values, while the other features utilize i values for the approximation of means. We utilize match+action tables populated with power of two values (1, 2, 4, 8, ...), along with appropriate binary masks to perform the match between i (or $i-1$) and the corresponding $prev\_pow(i)$ value. Lastly, the *online classifier* reads the value of additional metadata variables, containing each of the tree's votes. Afterwards, the module selects the classification with the highest number of votes. In order to evaluate the prototype, we insert a custom header, containing a field for the final classification.

To simplify the implementation of our prototype, we currently do not store multiple flows concurrently. In order to properly support the monitoring of multiple flows concurrently, it would be necessary to efficiently index flows into register positions. While it was not implemented by us, we know it is achievable by utilizing hash functions to im-

Table 5.1: Value registers and size in bits

Value	Length (bits)	Register	Metadata Variable
Destination Port	16	Yes	Header
Flow Duration	48	Yes	Yes
Packet Count	32	Yes	Yes
Header Length Sum	32	Yes	Yes
Initial Window	32	Yes	Yes
ACT Data Count	32	Yes	Yes
PSH Count	32	Yes	Yes
URG Count	32	Yes	Yes
IAT	48	Yes	Yes
IAT Total	48	Yes	Yes
IAT Minimum	48	Yes	Yes
IAT Maximum	48	Yes	Yes
IAT Mean	48	Yes	Yes
IAT A. Sum	48	Yes	Yes
Packet Length	32	No	Standard
Packet Length Total	32	Yes	Yes
Packet Length Minimum	32	Yes	Yes
Packet Length Maximum	32	Yes	Yes
Packet Length Mean	32	Yes	Yes
Packet Length A. Sum	32	Yes	Yes
Segment Size Total	32	Yes	Yes
Segment Size Minimum	32	Yes	Yes
Segment Size Maximum	32	Yes	Yes
Segment Size Mean	32	Yes	Yes
Segment Size A. Sum	32	Yes	Yes

Source: Author

plement a hash table (over registers). An example of a system that implements this logic is TurboFlow (SONCHACK, 2017).

5.2 Methodology

In order to better evaluate BACKORDERS, we utilized in our experiments the CICIDS2017 dataset (SHARAFALDIN; LASHKARI; GHORBANI, 2018). This dataset contains a large number of labeled flows, each one with over seventy calculated features. In particular, we utilized the subset of samples collected on Wednesday, July 5, 2017. On that day there were several DoS attacks that were generated, making this specific subset of higher interest for our purpose. For the subset in question, there were 692,703 flows generated and labeled. Out of the labeled flows, 440,031 were legitimate, corresponding to over 63.52% of the total number of flows. For the malicious flows, 5,796 were

DoS Slowloris attacks, 5,499 were DoS SlowHTTPTest attacks, 231,073 were DoS Hulk attacks, 10,293 were DoS GoldenEye attacks and only 11 were Heartbleed attacks. For our tests, we realized a binary division of classes, being classified as either legitimate or malicious, which includes every DoS and Heartbleed attacks.

First, we selected a subset from the set of available features. While the dataset contains over 70 features, we considered many of them to be too complex to be implemented in a programmable switch. Thus, we focused on selecting features that were viable to implement in a P4-enabled switch. The list of features implemented was shown in Table 4.3, and detailed in Subsection 4.3.3. Utilizing this subset of features we train a Random Forest model to be implemented in the data plane. As discussed in Subsection 2.3.1, the algorithm for induction of Classification Trees automatically tries to select the best features as early as possible in the process. Thus, we simply provided the list of available features to each RF, allowing the respective tree induction algorithm to detect and select the most significant features.

While offline classifiers tend to utilize dozens to hundreds of trees in a forest, a large number of learners would mean more processing time. Additionally, an excessive number of trees would require more match+action table entries, possibly exhausting the memory of the device. Along with the number of trees, the size of each tree should also be taken into account, as they impact the amount of match+action table entries utilized. As such, considering that the resources of a programmable switch are limited, we focused on exploring different configurations of meta-parameters for the RF, aiming to find a compromise between resource consumption and classification accuracy. In particular, we evaluated forests with 1, 3, 5, 7, and 9 trees, and trees with a maximum depth of 3, 4, 5, 6, and 7 layers.

Regardless of the meta-parameters utilized in each trained model, we also employed stratified K-fold cross-validation, with 5 folds per model. This evaluation technique divides the dataset into K folds, in this case in 5 folds. Then, we perform K iterations, where, in iteration i, we utilize fold i for evaluating and the remaining $K - 1$ folds for training. By dividing the samples in K folds while preserving the original dataset's class proportions, we ensure that in each step we evaluate the trained model with samples that were not used for training. Performing the evaluation with samples that were not used for training ensures that the results obtained better represent the trained model's performance with unknown data. Thus, in the case of the dataset utilized, which contains nearly 700,000 samples, approximately 560,000 ($\frac{4}{5}$, or 80%) samples are used in the training and the remaining nearly 140,000 ($\frac{1}{5}$, or 20%) samples are used for evaluating in each iteration of the K-fold algorithm. Each of the evaluated forests executes the K-fold algorithm, repeating the training and evaluation 5 times, with different training and evaluation subsets (folds) each time. To obtain the final score for each RF, we average the scores obtained in each of the 5 iterations.

For the evaluation of the different classifiers, we compare the F1-Score metric obtained by each classifier. This metric takes into account several aspects of the predictions realized. After this initial comparison, we select the models that have obtained high scores. For the selected learners, we compare further metrics, such as accuracy, precision, and recall. We consider these metrics to be relevant, as they allow us to compare the efficacy of different combinations of meta-parameters for the forests.

5.3 Results

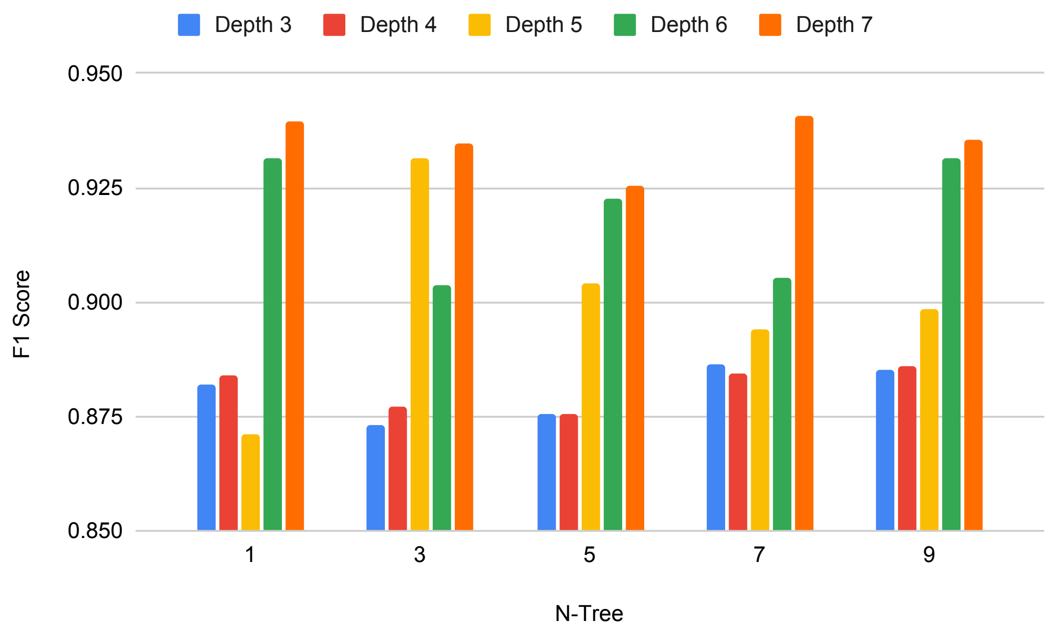
In this section, we present the results obtained in our experiments. In Subsection 5.3.1, we present and compare the scores obtained by each trained classifier. In Subsection 5.3.2, we present an analysis of the scalability of different RF configurations.

5.3.1 Learner Scores

In this subsection, we compare the different combinations of meta-parameters that define a Random Forest model. In particular, we present each of the model's obtained *F1-Score*, a popular metric for measuring the efficacy of ML learners. Additionally, considering legitimate traffic as the positive class, we also compare each model's *accuracy*, the proportion of samples that the classifier correctly predicted, *precision*, the proportion of the positive predictions that are correct, and *recall*, the proportion of samples with a positive class that was correctly predicted.

In Figure 5.1, we can visualize the F1-score obtained by each one of the trained RF models. As we can see in the figure, every model has an F1-Score higher than 0.85, including the forests with fewer trees and lower depth. However, by increasing the maximum depth, the trained forests obtain a considerably increased F1-score, even with a small number of classifiers per forest. Thus, even a small number of trees, while also limiting the maximum depth, can obtain reasonably good results.

Figure 5.1: F1 Score of different RF configurations



Source: Author

Next, we further compare a subset of the trained models, selecting only the models with high F1-score. As we have previously observed in Figure 5.1, out of the 25 models, the 10 following models have obtained high F1-score:

- 1 Tree, Depth 6;
- 1 Tree, Depth 7;
- 3 Trees, Depth 5;
- 3 Trees, Depth 6;
- 3 Trees, Depth 7;
- 5 Trees, Depth 5;
- 5 Trees, Depth 6;
- 5 Trees, Depth 7;
- 9 Trees, Depth 6;
- 9 Trees, Depth 7.

Thus, in Figure 5.2 we compare the accuracy, precision, and recall obtained by each of the selected models. As we can observe, the trained models have a tendency to have high precision. Considering that legitimate traffic is the positive class, this indicates that the model correctly classifies the majority of flows that it predicts as legitimate. While precision tends to be the highest metric achieved by the trained learners, the other metrics are not significantly lower.

5.3.2 Scalability Analysis

In this subsection, we present a theoretical analysis of the scalability of different configurations of Random Forests. We first present the worst-case scenario analysis of the processing and memory usage in terms of the number of trees in the forest and the maximum number of layers per tree. Next, we show some examples, based on the configurations presented in the previous subsection.

In terms of processing time in the data plane, the worst-case scenario will always be as many applications of tables as there are layers (maximum depth), multiplied by the number of trees. Thus, a forest with 5 trees, limiting each tree's maximum depth by 6, will, at worse, perform 5×6 comparisons, so 30 in this case. Ergo, the processing time is limited by $O(NM)$, where N is the number of trees and M is the maximum depth.

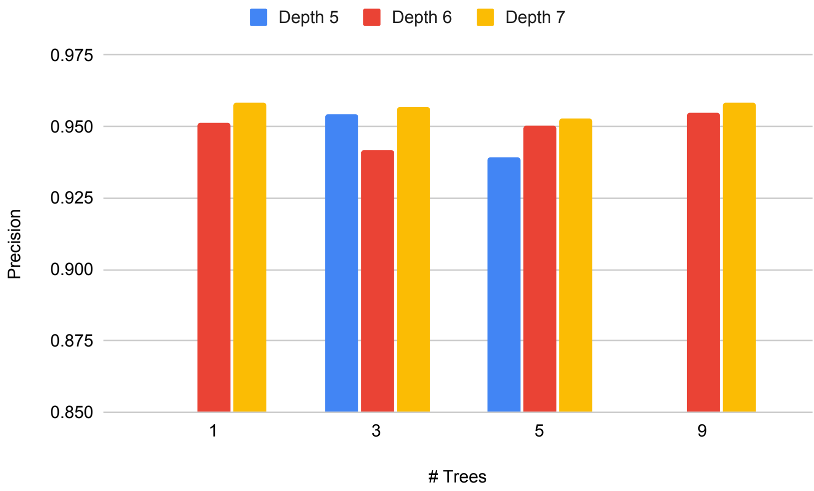
In terms of memory, each node is mapped into a single match+action table entry. Thus, a tree with 1 layer will have **1 node** (the root), while a tree with 2 full layers will have **3 nodes**, a tree with 3 full layers will have **7 nodes**, and so on. However, not every layer will have as many nodes as it can have, as trees generated by induction algorithms may have leaf nodes in any layer, not being limited to the last layer. Ergo, in the worst-case scenario, the memory usage by each tree is limited by $O(2^M)$, where M is the maximum depth of the tree. As we have N trees, the total memory utilization is limited by $O(N(2^M))$.

As an example, we utilize the 10 learners selected in the previous subsection. Recapitulating, in terms of the number of trees and the maximum depth, their configurations are:

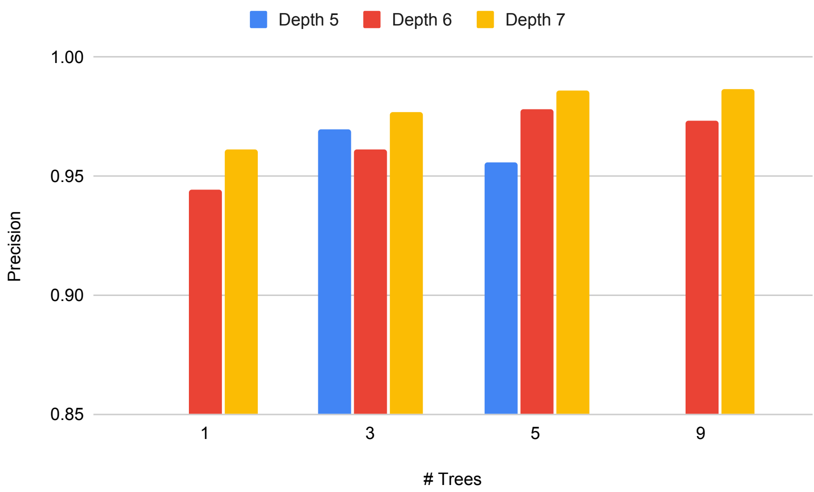
- 1 Tree, Depth 6;
- 1 Tree, Depth 7;
- 3 Trees, Depth 5;
- 3 Trees, Depth 6;
- 3 Trees, Depth 7;
- 5 Trees, Depth 5;
- 5 Trees, Depth 6;
- 5 Trees, Depth 7;

Figure 5.2: Comparison of different metrics between best trained models

(a) Accuracy



(b) Precision



(c) Recall

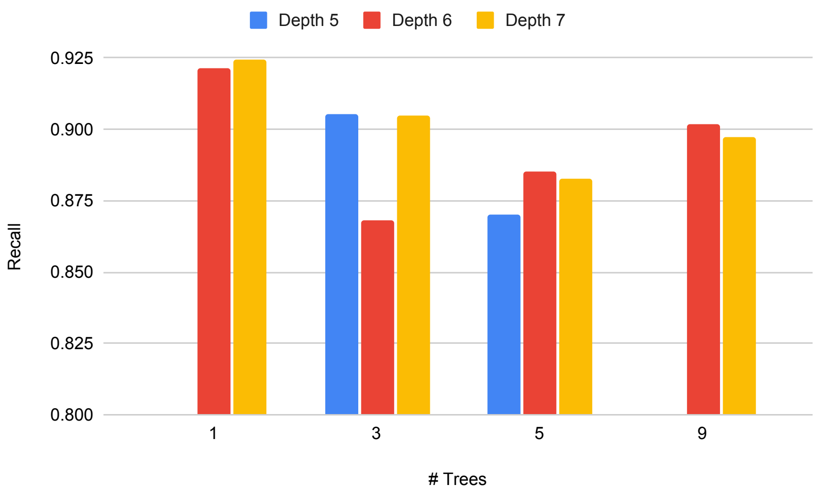


Table 5.2: Cost analysis of different RF configurations

# Trees	Max. Depth	Comparisons/tree	Total comparisons	Memory/tree	Total memory
1	6	6	6	63	63
	7	7	7	127	127
3	5	5	15	31	93
	6	6	18	63	189
	7	7	21	127	381
5	5	5	25	31	155
	6	6	30	63	315
	7	7	35	127	635
9	6	6	54	63	567
	7	7	63	127	1143

Source: Author

- 9 Trees, Depth 6;
- 9 Trees, Depth 7.

Thus, in Table 5.2 we calculate the number of comparisons and the memory usage, based on the worst-case scenario analysis presented previously. As previously explained, the number of comparisons per tree is limited, in the worst-case, by the maximum depth of the tree. In the case of a forest, the total cost is the sum of each tree's cost. As we are assuming every tree is a complete tree (full layers in every layer), it is simply one tree's cost multiplied by the number of trees. Analogously, the amount of match+action table entries, and thus memory, required by a tree is the number of nodes. For a forest, it's the sum of each tree's number of nodes.

5.4 Applicability

In this section, we discuss the real-world applicability of our solution. First, we discuss how well our approach deals with different types of DDoS attacks. Next, we discuss the life expectancy of trained forests.

As previously discussed in Section 2.1, there are multiple types of Denial of Service attacks. As the taxonomy of these attacks is not the main focus of this work, we simply categorized attacks in low-bandwidth, resource depletion attacks and high-bandwidth, flood attacks. The samples utilized in this work are the subset of samples collected on Wednesday, July 5, 2017, contained in the CICIDS2017 dataset (SHARAFALDIN; LASHKARI; GHORBANI, 2018). In that subset, the malicious flows exploited Application Layer vulnerabilities. Particularly, every attack was a low-bandwidth, resource depletion or security bug attack. Our work focuses on the detection of these types of attacks, as some traditional approaches have difficulty in detecting them.

However, BACKORDERS utilizes a Random Forest to perform the classification of flows based on several statistical features, such as inter-arrival-time of packets, length of packets, and length of packet headers. Thus, our approach may not be appropriate for the detection of high-bandwidth, flooding attacks, such as SYN Flood attacks and ICMP Ping Flood attacks. We believe other approaches may be more suitable for the detection of these types of attacks. An example is the system proposed by Lapolli, Marques, and Gaspary (LAPOLLI; MARQUES; GASPARY, 2019). As previously mentioned in Section 3.1, their work calculates the entropy of IP addresses to detect potential DDoS attacks. It might be possible to combine both systems, providing detection for both high-bandwidth and low-bandwidth attacks.

Despite being more suited for the detection of a particular class of DoS attacks, our system achieved high accuracy in the tests realized, as presented in Subsection 5.3.1. However, as briefly mentioned in Subsection 4.3.1, supervised Machine Learning models require a high number of previously labeled samples. Additionally, these samples must be able to generalize the characteristics of data that was not present in the training set. Thus, the dataset utilized to train the RF inserted into the data plane must be able to appropriately generalize real-world network traffic. However, real-world network traffic is not static, that is, it changes over time. For instance, the proportion of encrypted Internet traffic started increasing over the years. Thus, techniques that required access to unencrypted payloads started losing effectiveness.

In order to conform to the changes in network traffic profile, real-world deployments of a more robust version of BACKORDERS would require the periodic training of updated Random Forest models. While ideally this would happen frequently, the collection and (possibly offline) classification of network flow is a difficult task, as briefly mentioned in Subsection 4.3.1. Thus, we believe there needs to be further research on the topic of the life expectancy of trained models of RF, keeping in mind the specific goal of classifying network flows and the dynamic nature of the Internet.

Although the exact life expectancy of RF models trained for the classification of network flow is unknown, our current prototype allows for the reconfiguration of the Random Forest inserted in the data plane without incurring downtime to the programmable switch. Particularly, by utilizing match+action tables to store the nodes, it is possible to reconfigure these tables to accommodate a new forest. However, as the number of tables depends on the number of trees and the maximum depth of each tree, new forests are limited by these parameters. In order to insert a model with different parameters, the device would need to be turned off and then re-programmed with updated P4 source code.

6 CONCLUSION

In this chapter, we present the conclusion of this work. In Section 6.1, we present a summary of the contributions of this work. In Section 6.2, we discuss future work.

6.1 Summary of Contributions

In this work, BACKORDERS (distriButed deniAl of serviCe attacK detectOr using RanDom forEst in pRogrammable Switches), a system for classifying network flow in programmable data planes was presented. The system implements a Random Forest classifier, mapping its structure to fit in a P4-enabled switch. This process takes the information contained in each node and maps it into match+action tables. By mapping the data structure that tends to be recursive into a set of entries of a table, we manage to perform the evaluation of nodes sequentially. Once the learner is mapped into the programmable data plane, we calculate the features of each flow being monitored. These features are later utilized by the Classification Trees in order to provide a classification of the network flow, identifying whether it is a legitimate or malicious flow. By utilizing match+action tables, a structure that is highly optimized for programmable switches, we efficiently perform the classification of a sample by utilizing a previously trained Random Forest.

We believe that this work shows that it is possible to utilize machine learning models in the data plane, performing predictions with the high accuracy that ML learners are known for, while also meeting the restrictions imposed by the limited hardware and short processing time required by programmable switches.

Additionally, we propose, implement, and evaluate an algorithm for approximating mean values in the data plane. Despite utilizing only additions, subtractions, and bit-shifts, our algorithm achieves an approximation that is very close to the exact value. Thus, systems that depend on mean values as a part of their decision logic have an alternative to utilizing exponentially weighted moving averages as a substitute for moving averages.

6.2 Future Work

In future work, we aim to optimize some aspects of our system. In specific, in order to perform the classification of multiple network flows concurrently, we must minimize the amount of memory utilized by each flow. The amount of memory utilized by our currently implemented features can be too high, as analyzed in Section 5.1. In order to fit a set of features for each flow in a programmable switch, considering that the usual numbers of flows are in the dozens to hundreds of thousands, each feature must take a minimal amount of memory. Thus, we believe that there are techniques that can be employed to reduce the number of bits of multiple features of each flow.

In order to further minimize the memory utilized by our system, in future work, we plan to implement the generation of code for the *feature extractor* module by using a smarter approach. Rather than utilizing a register and metadata for every available feature, we will only include registers, metadata variables, and actions for features that were utilized by the inserted RF. This approach would remove some registers for features that were not utilized by the forest, lowering the consumption of memory. As a drawback, however, it would difficult the insertion of a new forest in the data plane while the P4-enabled switch is operational. Thus, in future work, we plan on evaluating the trade-off between our currently implemented approach and this proposed optimization.

As mentioned in Section 5.1, our prototype does not currently support concurrent flows. Regardless of the maximum number of flows that can be monitored concurrently, we plan on implementing hash tables in the future. With hash tables, we can remove the limitation of a single flow at a time, even if the maximum number of supported concurrent flows is not sufficiently high at first.

Additionally, we would like to explore different features, in order to attempt to find features that better aid in the classification of network traffic. While the dataset we utilized in our evaluation provides a high number of features, many of them are redundant, while some of the others are extremely complex to be implemented in a P4-enabled switch. Thus, we plan on attempting to implement the extraction of different features. The area of feature selection has seen intense research activity in the last few years (SANTOS DA SILVA et al., 2015). With better features, it might be possible to train smaller trees, while retaining or increasing the accuracy of classification.

REFERENCES

AKAMAI. **State of the Internet / security: Web Attacks**. 2018. Available from Internet: <https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-summer-2018-web-attack-report.pdf>. Accessed in: 2020-11-03.

ANTONELLO, R. et al. Deep packet inspection tools and techniques in commodity platforms: Challenges and trends. **Journal of Network and Computer Applications**, v. 35, p. 1863–1878, 11 2012.

ASOSHEH, A.; IVAKI, N. A comprehensive taxonomy of ddos attacks and defense mechanism applying in a smart classification. **WSEAS Transactions on Computers**, v. 7, p. 281–290, 04 2008.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656890>.

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, Oct 2001. ISSN 1573-0565. Available from Internet: <https://doi.org/10.1023/A:1010933404324>.

BREIMAN, L. et al. Classification and regression trees. In: . [S.l.: s.n.], 1983.

BUDIU, M.; DODD, C. The p416 programming language. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 1, p. 5–14, sep. 2017. ISSN 0163-5980. Available from Internet: <https://doi.org/10.1145/3139645.3139648>.

BUSSE-GRAWITZ, C. et al. pforest: In-network inference with random forests. **ArXiv**, abs/1909.05680, 2019.

DONG, S.; ABBAS, K.; JAIN, A. A survey on distributed denial of service (ddos) attacks in sdn and cloud computing environments. **IEEE Access**, v. 7, p. 80813–80828, 2019.

DURAVKIN, I.; LOKTIONOVA, A.; CARLSSON, A. Method of slow-attack detection. In: **2014 First International Scientific-Practical Conference Problems of Infocommunications Science and Technology**. [S.l.: s.n.], 2014. p. 171–172.

FEBRO, A.; XIAO, H.; SPRING, J. Distributed sip ddos defense with p4. In: **2019 IEEE Wireless Communications and Networking Conference (WCNC)**. [S.l.: s.n.], 2019. p. 1–8.

FINSTERBUSCH, M. et al. A survey of payload-based traffic classification approaches. **IEEE Communications Surveys & Tutorials**, PP, p. 1–22, 12 2013.

HAAHR, M. **RANDOM.ORG - Gaussian Random Number Generator**. 2020. Available from Internet: <https://www.random.org/gaussian-distributions/>. Accessed in: 2020-11-17.

HO, T. K. Random decision forests. In: **Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1**. USA: IEEE Computer Society, 1995. (ICDAR ’95), p. 278. ISBN 0818671289.

ILIYASU, A.; DENG, H. Semi-supervised encrypted traffic classification with deep convolutional generative adversarial networks. **IEEE Access**, PP, p. 1–1, 12 2019.

JORDAN, S. Some traffic management practices are unreasonable. In: . [S.l.: s.n.], 2009. p. 1 – 6.

KUNCHEVA, L. I. **Combining Pattern Classifiers: Methods and Algorithms**. USA: Wiley-Interscience, 2004. ISBN 0471210781.

KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach (6th Edition)**. 6th. ed. [S.l.]: Pearson, 2012. ISBN 0132856204.

LAPOLLI, Â. C.; MARQUES, J.; GASPARY, L. Offloading real-time ddos attack detection to programmable data planes. **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**, p. 19–27, 2019.

LASHKARI, A. et al. **ahlashkari/CICFlowMeter**. 2020. Available from Internet: <https://github.com/ahlashkari/CICFlowMeter>. Accessed in: 2020-11-17.

LI, Y. et al. Accelerating distributed reinforcement learning with in-switch computing. In: **Proceedings of the 46th International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2019. (ISCA '19), p. 279–291. ISBN 9781450366694. Available from Internet: <https://doi.org/10.1145/3307650.3322259>.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1355734.1355746>.

MIRKOVIC, J.; REIHER, P. A taxonomy of ddos attack and ddos defense mechanisms. **ACM SIGCOMM Computer Communication Review**, v. 34, 05 2004.

MOORE, D. et al. Inferring internet denial-of-service activity. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 24, n. 2, p. 115–139, may 2006. ISSN 0734-2071. Available from Internet: <https://doi.org/10.1145/1132026.1132027>.

MUSUMECI, F. et al. Machine-learning-assisted ddos attack detection with p4 language. In: **ICC 2020 - 2020 IEEE International Conference on Communications (ICC)**. [S.l.: s.n.], 2020. p. 1–6.

NOSSENSON, R.; POLACHECK, S. On-line flows classification of video streaming applications. In: . [S.l.: s.n.], 2015. p. 251–258.

POLIKAR, R. Polikar, r.: Ensemble based systems in decision making. ieee circuit syst. mag. 6, 21-45. **Circuits and Systems Magazine, IEEE**, v. 6, p. 21 – 45, 10 2006.

QUINLAN, J. R. Simplifying decision trees. **Int. J. Man-Mach. Stud.**, Academic Press Ltd., GBR, v. 27, n. 3, p. 221–234, sep. 1987. ISSN 0020-7373. Available from Internet: <https://doi.org/10.1016/S0020-7373(87)80053-6>.

QUINLAN, J. R. **C4.5: Programs for Machine Learning**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602380.

SANTOS DA SILVA, A. et al. Identification and selection of flow features for accurate traffic classification in sdn. In: **2015 IEEE 14th International Symposium on Network Computing and Applications**. [S.l.: s.n.], 2015. p. 134–141.

SANVITO, D.; SIRACUSANO, G.; BIFULCO, R. Can the network be the ai accelerator? In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. New York, NY, USA: Association for Computing Machinery, 2018. (NetCompute ’18), p. 20–25. ISBN 9781450359085. Available from Internet: <https://doi.org/10.1145/3229591.3229594>.

SAPIO, A. et al. **Scaling Distributed Machine Learning with In-Network Aggregation**. 2019.

SAUERESSIG, M. **p4features**. 2020. Available from Internet: <https://github.com/LaironSk/p4features>. Accessed in: 2020-11-08.

SHARAFALDIN, I.; LASHKARI, A. H.; GHORBANI, A. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: . [S.l.: s.n.], 2018. p. 108–116.

SHARAFALDIN, I. et al. Developing realistic distributed denial of service (ddos) attack dataset and taxonomy. In: . [S.l.: s.n.], 2019. p. 1–8.

SIKORA, M. et al. Design of advanced slow denial of service attack generator. **2020 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)**, p. 99–104, 2020.

SIMSEK, G. et al. Dropppp: A p4 approach to mitigating dos attacks in sdn. In: YOU, I. (Ed.). **Information Security Applications**. Cham: Springer International Publishing, 2020. p. 55–66. ISBN 978-3-030-39303-8.

SIRACUSANO, G.; BIFULCO, R. In-network neural networks. **CoRR**, abs/1801.05731, 01 2018.

SONCHACK, J. **jsonch/turboflow**. 2017. Available from Internet: <https://github.com/jsonch/p4_code/tree/master/netronome/turboflow>. Accessed in: 2020-11-26.

WOLFE, S. **Amazon’s one hour of downtime on Prime Day may have cost it up to $100 million in lost sales**. Business Insider, 2018. Available from Internet: <https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7>. Accessed in: 2020-11-03.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets ’19), p. 25–33. ISBN 9781450370202. Available from Internet: <https://doi.org/10.1145/3365609.3365864>.

YAN, Q. et al. Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges. **IEEE Communications Surveys & Tutorials**, v. 18, p. 1–1, 10 2015.

ZARGAR, S. T.; JOSHI, J.; TIPPER, D. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. **IEEE Communications Surveys & Tutorials**, v. 15, p. 2046 – 2069, 11 2013.