Introducing Packet-Level Analysis in Programmable Data Planes to Advance Network Intrusion Detection

Roberto Doriguzzi-Corin[α], Luis Augusto Dias Knob[α], Luca Mendozzi[β], Domenico Siracusa[α], Marco Savi[β]
[α]Cybersecurity Centre, Fondazione Bruno Kessler, Trento - Italy
[β]University of Milano-Bicocca, Department of Informatics, Systems and Communication (DISCo), Milano - Italy

Abstract

Programmable data planes offer precise control over the low-level processing steps applied to network packets, serving as a valuable tool for analysing malicious flows in the field of intrusion detection. Albeit with limitations on physical resources and capabilities, they allow for the efficient extraction of detailed traffic information, which can then be utilised by Machine Learning (ML) algorithms responsible for identifying security threats. In addressing resource constraints, existing solutions in the literature rely on compressing network data through the collection of statistical traffic features in the data plane. While this compression saves memory resources in switches and minimises the burden on the control channel between the data and the control plane, it also results in a loss of information available to the Network Intrusion Detection System (NIDS), limiting access to packet payload, categorical features, and the semantic understanding of network communications, such as the behaviour of packets within traffic flows. This paper proposes P4DDLe, a framework that exploits the flexibility of P4-based programmable data planes for packet-level feature extraction and pre-processing. P4DDLe leverages the programmable data plane to extract raw packet features from the network traffic, categorical features included, and to organise them in a way that the semantics of traffic flows are preserved. To minimise memory and control channel overheads, P4DDLe selectively processes and filters packet-level data, so that only the features required by the NIDS are collected. The experimental evaluation with recent Distributed Denial of Service (DDoS) attack data demonstrates that the proposed approach is very efficient in collecting compact and high-quality representations of network flows, ensuring precise detection of DDoS attacks.

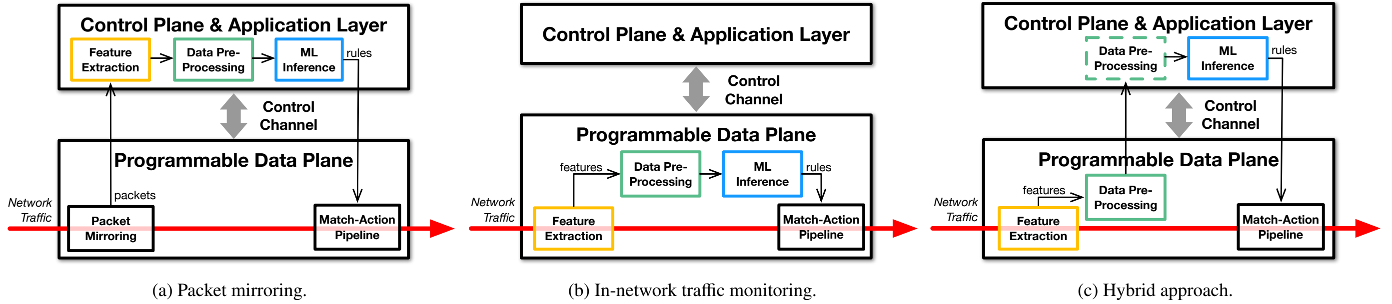
Keywords: Programmable Data Planes, Network Intrusion Detection, Packet-Level features

1. Introduction

Network intrusions and anomalies are one of the most significant plagues in modern communication networks. As the number and complexity of incidents are constantly increasing [1], it has become imperative to implement meticulous monitoring and robust counteraction measures to effectively detect and mitigate these threats. In the past decade, network monitoring has lived a second youth, thanks in large part to the prominence of Software Defined Networking (SDN) [2, 3] and to the rise of ML technologies in networking [4]. Within the network security domain, the fusion between the centralised control plane of SDN and data-driven threat detection powered by ML algorithms has proved remarkable efficacy in promptly identifying and mitigating network intrusions and anomalies [5].

Despite the undeniable benefits brought by the synergy between SDN and ML, the implementation of robust network security solutions remains a challenge, primarily due to the fine-grained traffic features required by ML algorithms. In this context, a direct and effective approach for leveraging the centralised control plane of SDN, while ensuring that ML algorithms receive the necessary traffic information, is through a technique known as *packet mirroring* [6]. By employing packet mirroring, a duplicate copy of the network traffic is transmitted from the data plane to the control plane, where it undergoes traffic feature extraction and pre-processing before ML algorithms can be executed upon it (Figure 1a). Unfortunately, the channel between data and control planes often comes with severe bandwidth and latency bottlenecks [7, 8, 9, 10]. These inherent limitations imply that it may be overwhelmed by the sheer volume of mirrored data [11], thereby jeopardising ordinary SDN network operations and compromising prompt response to ongoing network attacks.

The recent emergence of programmable data planes [12] has introduced a technology that offers a promising solution to tackle the above challenges. Programmable data planes enable the customization of the data plane pipeline (known as *fastpath* [13]) using domain-specific languages like P4 [14]. This level of programmability and flexibility empowers network practitioners to optimise feature extraction, data pre-processing, and ML inference operations, which can be offloaded to the data plane (see Figure 1b): with programmable data planes, it thus becomes possible to finely manipulate the type and volume of traffic data forwarded to the control plane (known as *slowpath* [13]). However, it is important to note that the data plane has inherent limitations, such as a limited set of available arithmetic instructions and memory capacity in the match-action tables [15]. As a result, only simple ML models can be effectively offloaded, which may lead to noticeable degradation in infer-



(a) Packet mirroring.

(b) In-network traffic monitoring.

(c) Hybrid approach.

Figure 1: Approaches to traffic monitoring with programmable data planes.

ence performance [16].

We advocate leveraging the full potential of the centralised SDN control plane by executing ML inference directly on the SDN controller. This approach allows network monitoring operations to exploit the advantages of a comprehensive, global view of the network, while also enabling the implementation and execution of sophisticated ML algorithms. By performing ML inference on the controller, the network can benefit from enhanced analysis and decision-making capabilities, leveraging the rich information available at the control plane. Simultaneously, we recognise the value of programmable data planes in performing traffic feature extraction and pre-processing. By leveraging the programmability of data planes, it becomes possible to precisely control the amount of data that traverses the control channel. This approach allows for selective processing and filtering of data at a data plane level, reducing the burden on the control channel and enabling efficient resource allocation (see Figure 1c).

This concept has been extensively explored in the scientific literature, with several studies proposing solutions based on the aggregation of traffic statistics computed within the data plane. The primary objective of these solutions is to optimise the load on the control channel and effectively manage memory utilization in the data plane [17, 18, 19, 20, 21]. One notable drawback of relying solely on statistical traffic features, such as flow duration, averages, maximums, minimums, and standard deviations of attributes like packet size, rate, and inter-arrival time, is the inability to report categorical features (e.g., IP and TCP flags, Differentiated Services Code Point (DSCP), etc.), the timestamp of packets or portion of packets' payload directly to the control plane. These packet-level traffic features are crucial to detect some types of network intrusions, such as brute force attacks (e.g., by monitoring the TCP flags, including SYN, ACK, FIN and RST flags [22]), anomalous QoS settings (monitoring the DSCP code points) [23], TCP, UDP and ICMP fragmentation attacks (monitoring the IP flags) [24], etc. In addition, the timestamp of packets and portions of the payload are employed in combination with categorical features by certain ML-based NIDSs in the current state of the art (e.g., [25, 26, 27, 28, 29], among many others) to learn the semantic of malicious traffic flows and to segregate them from legitimate network activities.

In this paper, we propose P4DDLe (P4-empowered Ddos detection with Deep Learning), a framework for efficient packet-level feature extraction and pre-processing in P4-based programmable data planes. P4DDLe inherently supports the categorical features, and it has been designed to preserve the semantic integrity of the flows. With these properties, P4DDLe ensures that the ML-based NIDS executed in the control plane can capture the relevant protocols, data formats, application behaviours and traffic patterns, allowing for a comprehensive understanding of the network traffic semantics.

P4DDLe exploits stateful memory objects (*P4 registers*) and a *counting Bloom filter* to efficiently select and store the packet-level features that are relevant for the ML model. This approach enables the data plane to discard redundant data, which would otherwise be disregarded by the control plane, with great benefits in terms of control channel usage and control plane processing load. The selected features are stored within two ring buffers, which have been specifically designed to avoid race conditions between the control plane (reading the stored features) and the data plane (writing them). The latter aspect has been often neglected by previous works.

To the best of our knowledge, this is the first work addressing the challenges of packet-level feature extraction in the data plane for network security applications. In contrast to existing works, P4DDLe combines the flexibility of packet-level features with the network-wide view provided to the NIDS by the centralised SDN control plane (not available when the NIDS runs completely in the data plane [30, 31]). We believe that the combination of these two features sets P4DDLe apart from current state-of-the-art approaches to network attack detection with programmable data planes.

We have extensively evaluated P4DDLe using a recent dataset of DDoS attacks and a well-known NIDS based on a Convolutional Neural Network (CNN) that takes packet-level features as input [25]. We have compared our approach against a naïve strategy for packet-level feature extraction, which stores the data in the buffers sequentially without any optimisation logic. In a comprehensive range of testing scenarios, we empirically show that P4DDLe is able to collect more feature-rich traffic flows. As a result, our approach achieves a significantly lower system False Negative Rate (FNR), measured as the sum of overlooked and misclassified malicious flows, enabling faster detection and mitigation of network attacks.

The main contributions of this work can be summarised as follows:

1. A framework for P4-based programmable data planes called P4DDLe, designed to cope with the requirements of modern NIDSs, which often rely on raw packet-level features for accurate cyber threat detection.

2. A mechanism based on counting Bloom filters to efficiently group packet-level features into traffic flows while minimizing the chances of collisions between the flow identifiers.

3. A comprehensive assessment of the proposed approach, including a sensitivity analysis of the Bloom filter size (directly linked to the probability of collisions), and the evaluation of a state-of-the-art NIDS in detecting realistic DDoS attacks using P4DDLe's packet-level representation of the network traffic.

The remainder of the paper is structured as follows. Section 2 reviews and discusses the related work. Section 3 provides an overview of P4 data plane programming, counting Bloom filters and the CNN used in our experiments. Section 4 presents P4DDLe's architecture and the methods for feature extraction and storage. Section 5 details the setup of the experimental evaluation presented in Section 6. Finally, the conclusions are given in Section 7.

2. Related Work

One of the primary challenges in network traffic monitoring is finding a balance between the accuracy of the monitored traffic attributes and the level of resources (both network and computing) required to achieve it. In this regard, programmable network devices can be exploited to handle the monitoring operations in the data plane, effectively reducing the burden on both control plane and control channel. Nevertheless, despite the adaptability of modern data plane implementations, including those relying on the Programming Protocol-independent Packet Processors (P4) programming language, they still have certain limitations in the types of operations they can perform. To address this issue, a common technique is to offload some of the computation from the data plane to the control plane (e.g., performing ML inference), where more computing power and advanced tools are accessible. Nevertheless, this approach could be constrained by the limited capacity of the communication/control channel linking the data and control planes.

This section provides an overview of recent research that has tackled these challenges, with a particular emphasis on the network security domain, where achieving a balance between resource utilization and monitoring accuracy is critical for quickly identifying and responding to cyber threats.

2.1. ML-based In-network Traffic Classification

In network monitoring applications, an effective solution to prevent the control channel from being overloaded with network data is to offload the tasks of traffic feature extraction, data pre-processing, and ML inference to the programmable data plane (see Figure 1b). This approach, also known as the *in-network* approach, offers several advantages in terms of control channel utilization by minimising the interaction between data and control planes. However, despite its benefits, this approach also presents some significant drawbacks that should be taken into account.

It is worth noting that the P4 programming language does not support floating-point operations and divisions [32]. Consequently, some highly effective ML algorithms, including Artificial Neural Networks (ANNs), cannot be directly implemented in the data plane, as those algorithms rely on model weights that are usually represented as floating-point numbers. Recent initiatives have been trying to enable floating-point operations to P4, either with the adoption of dedicated hardware Floating Point Units (FPUs) [33] or by proposing hardware changes to the data plane's architecture of programmable devices [34]. Although promising, these approaches present limited portability to existing P4 devices (switches and/or smartNICs), and may require a non-negligible time horizon for their adoption in more recent products.

For this reason, most of the existing proposals for ML-based in-network traffic classification that exploit programmable data planes adopt simple ML models, such as decision trees [16][35][36][37][38], binary decision trees [39], random forests [40][41][37][30][38], Support Vector Machine (SVM) [16][37], Naïve Bayes [16][37], K-means [16][37], XGBoost [37]. Concerning Neural Network (NN) models, Binary Neural Network (BNN) can be successfully offloaded [31][42] thanks to their simplicity, as model weights are binary values.

In all these works, the ML models are implemented by exploiting either match-action tables, populated by the control plane with appropriate rules, or by making use of P4 registers, i.e., by hard-coding the model into the P4 program. The former case is more flexible, as the model can be updated by the control plane at runtime (e.g. after re-training) by just injecting new rules, but it consumes memory that is typically dedicated to traffic forwarding. In a recent work, Razavi et al. [43] have implemented an ANNs directly in the data plane. A notable limitation of this approach is the encoding of floating-point weights as 8-bit integers. As no performance evaluation is presented, the efficacy of the proposed solution remains uncertain. In addition, some works [44][45] focus on the design and implementation of frameworks for offloading to the data plane the most appropriate ML model according to the task to be executed.

In general, what we can conclude by analysing the aforementioned papers is that the performance of ML-based in-network traffic classification is often hindered by the inherent limitations of the P4 language and/or of the underlying hardware. This can make it challenging to offload ML models with satisfactory classification performance.

In contrast to the solutions mentioned above, we have opted to execute the traffic classifier in the control plane. This approach allows us to choose the most appropriate ML model for a given task, without being constrained by the limitations of the P4 language or the hardware of the devices. By leveraging the flexibility of the control plane, we can use more advanced ML models to achieve better accuracy and performance as needed. Furthermore, the centralised nature of the SDN control plane

enables the NIDS to leverage network-wide traffic data, facilitating more accurate and comprehensive detection of network threats. This advantage is not attainable with the distributed inference employed by *in-network* approaches.

2.2. Interaction between Control Plane and Data Plane

Offloading a portion of traffic processing to the control plane (Figure 1c) requires careful consideration of the limitations imposed by the control channel in terms of bandwidth and latency. This consideration becomes even more crucial when a substantial amount of data must be exchanged between the two planes, as is for instance required when executing ML inference in the control plane for DDoS attack detection [25].

NetWarden [13] is a defense solution designed to mitigate network covert channels, utilising programmable data planes. In order to achieve this goal, NetWarden's approach involves restricting the data plane to performing per-packet operations exclusively on header fields. On the other hand, the control plane is only used for batch operations, reducing the need for interaction between the control and data planes and optimising the overall performance of the system. DySO [10] is a monitoring framework that shares the same fundamental principles as NetWarden, seeking to eliminate the bottleneck that can occur between data and control planes. Our research similarly aims to minimise the interaction between the two planes, accomplishing this by conducting feature extraction and selection in the data plane. By only forwarding essential data to the control plane for input to the ML model, we can significantly reduce the overall system latency and optimise performance.

A couple of recent works take different approaches to solve the bottleneck issue with the control channel. Escala [9] operates under the assumption that multiple control channels may be in use and may become overloaded. To address this issue, Escala offers the ability to elastically scale these channels at runtime as necessary, and to seamlessly migrate event streams (i.e., data transmitted from the data to the control plane) between different channels. Chen et al. [8] introduce MTP, a novel framework designed to optimise the placement of measurement points in the data plane, with the aim of minimising the overall cost associated with monitoring servers and network devices. This framework is based on an Integer Linear Programming (ILP) formulation, which defines a constraint on the capacity of the links between the data and control planes. In general, these works are orthogonal to our proposal and could be adopted to further alleviate the performance bottleneck of the control channel.

Finally, Bermudez Serna et al. [46] focus on security aspects. They propose a reactive configuration mechanism that can be adopted to counteract attacks aimed at overloading the control channel which, given its constrained capacity, could easily get saturated by malicious traffic. Also, this strategy is orthogonal to our proposal and could be adopted together with it for enhanced security.

2.3. Efficient Feature Extraction in the Data Plane

Efficient feature extraction in programmable data planes is a challenging problem that has received limited attention in the scientific literature. The goal is to generate features that are compatible with the input layer of a ML classifier, which is executed in the control plane for traffic monitoring or classification. Despite the importance of this problem, there are only a few studies that have addressed it and most of them, like our work, focus on DDoS detection as a use case.

In this respect, FlowLens [19] is a flow classifier designed for programmable switches that implements a mechanism for optimising the amount of data to be stored in memory and transferred through the control channel. While this feature enables efficient processing of network flows, race conditions on the shared memory are managed by the control plane, which deletes the flow tables to avoid concurrent read/write operations on the registers that store the traffic features. According to the paper, the flow tables are left empty until the reading process is completed. As a result, any incoming packets during this period cannot be processed or collected.

Musumeci et al. [17] propose an approach to SYN Flood DDoS attack detection that leverages P4-based programmable data planes for feature extraction and pre-processing. The P4 program extracts statistical traffic features computed over a predefined time window, such as average packet length, percentage of TCP and UDP packets and percentage of TCP packets with active SYN flag. A similar approach is adopted by Zang et al. [20], who propose a DDoS detection system based on ensemble learning and data-plane feature extraction and pre-processing. Due to the reliance on global statistical features rather than per-flow features, both approaches are limited to producing binary outputs indicating the presence of an ongoing attack. However, their ML classifiers cannot identify specific malicious flows. HybridDAD [21] is another solution based on statistical features. In this case, the output of the ML algorithm is a label that indicates whether there has been an attack during the previous time window, plus the class of the attack (among four types of DDoS flood attacks).

ORACLE [18] implements a flow-based feature extraction and pre-processing in the data plane for the detection of DDoS attacks, which is offloaded to ML algorithms executed in the control plane. To accomplish this, ORACLE employs a P4 program that collects per-flow statistical features, including flow duration, standard deviation of inter-arrival time, average packet size, and standard deviation of packet length. Apart from the inherent complexity of calculating statistical features in the data plane due to the limitations of the P4 language, ORACLE utilises hashing to index flows. With this approach, packets from different flows can be grouped together in the case of collisions. As a result, the computed statistics may be incorrect, making the final flow statistics unreliable and unusable.

Our solution addresses the limitations of prior research by implementing a double-block storage mechanism in the data plane, which effectively prevents race conditions. We achieve this by using two separate blocks (implemented using a set of P4 registers) and switching between them using a dedicated P4 register. The data plane writes to one block while the control plane reads from the other, ensuring consistency and minimising conflicts. Compared to previous methods based on statis-

tical flow-level features, our approach leverages a packet-level representation of network traffic, which preserves the semantics of flows (i.e., the behaviour of packets within each flow) and the categorical features of packets (e.g., TCP flags, ICMP type, etc.). This information is crucial for various ML-based NIDSs [25, 26, 27, 28, 29]. On the other hand, packet-level features may entail a greater amount of data transmission via the control channel, compared to flow-level statistical features. We alleviate this by proposing a *flow-based* storage mechanism, that relies on a statistical technique called *counting Bloom filter* [47]. By doing so, we can keep in memory only the essential data required for control plane operations, reducing the need to transmit extra traffic information that would ultimately be discarded.

3. Background

3.1. Data Plane Programming with P4

P4 is a language for expressing how the network traffic is processed by the data plane of programmable network elements such as hardware and software switches. P4 is target-independent, thus it supports a variety of targets such as ASICs, FPGAs, NICs and software switches.

The basic components of a P4-programmable forwarding pipeline are illustrated in Figure 2. This diagram outlines the *Version 1.0 Switch Architecture Model* [48], abbreviated as *V1model*, which is widely adopted by developers for P4 program implementation. We utilised this model as a reference for P4DDLe. The pipeline encompasses a *Parser*, a state machine that extracts packet headers and metadata from the incoming bitstream, a *Checksum verification* control block, an *Ingress Match-Action* processing control block, an *Egress Match-Action* processing control block, a *Checksum update* control block and a *Deparser*, which assembles the headers with the payload received from the Parser.

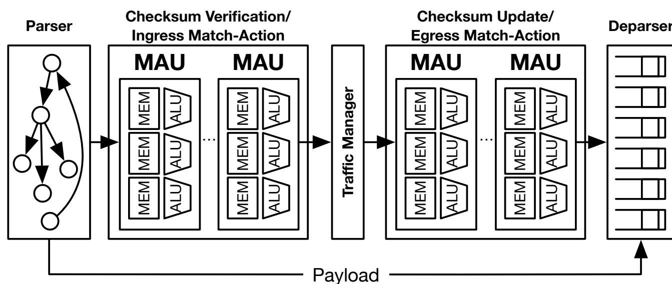


Figure 2: P4-programmable pipeline components (*V1model*).

As shown in the figure, the Match-Action block may consist of one or multiple Match-Action Units (MAUs). Each MAU comprises TCAM and SRAM memory blocks (MEM in the figure) for various purposes such as match-action tables, registers, meters, etc. The Arithmetic Logic Units (ALUs) blocks instead execute arithmetic operations and modifications on the packets' headers and metadata based on the content of the match-action tables, which host the traffic forwarding rules. ALUs may use stateful objects stored in the SRAM memory (e.g., registers) for additional operations.

Relevant to this work are the stateful objects of type *register*. Registers belong to a wider category of stateful elements called *extern objects*, which also include counters and meters. Unlike match-action tables, which are stored in the TCAM memory and can be modified only by the control plane, extern objects in the SRAM memory can be read and written from both data and control planes. Therefore, the access to registers, and to extern objects in general, should be managed with atomic operations to prevent race conditions.

In this work, we use registers to store per-packet and per-flow data that is written by the data plane and periodically read by the control plane. We initially enclosed read/write operations into atomic blocks to avoid race conditions between data and control planes. However, as we noticed that this approach causes a considerable loss of traffic data when the control plane locks the registers for reading, we adopted a more efficient strategy consisting of using two different registers, one for writing and one for reading. A third register serves as a switch to orchestrate the access to those two registers. Section 4 provides more details on this mechanism.

3.2. Counting Bloom filters

A *counting Bloom filter* [47] is an array of m cells that utilizes probabilistic hashing techniques to keep an approximate count of items. We use a counting Bloom filter to efficiently count the number of packets belonging to the same flow that have been collected in a specific time frame. The key idea behind a counting Bloom filter is to use h hash functions to map the elements of a set onto an array of size m. To achieve this, each element is hashed h times using different hash functions. The resulting hash values are used to index into the array, and the corresponding array cells are incremented by one. Since multiple elements can map to the same cell due to collisions, the counters of an element may not be incremented equally. To estimate the count of an element, the minimum value of the cells to which the element is mapped is used. The rationale behind this is that the minimum count is less likely to have been affected by collisions with other elements.

This is a highly efficient method for counting the number of packets per flow (or packets/flow) that have been collected in a memory block within a given time window. While a single hash function, such as CRC32, can also perform this task, counting Bloom filters are less prone to collisions, which can lead to inaccuracies in the count.

3.3. Neural network architecture

To validate our approach to data plane traffic feature extraction, we consider the DDoS attack detection use case. In particular, we focus on volumetric DDoS attacks, as they are particularly challenging to handle in constrained systems, such as network switches and SmartNICs, due to the large amounts of data rate such attacks can produce.

In this work, we adopt a state-of-the-art solution for DDoS attack detection called Lightweight, Usable CNN in DDoS Detection (Lucid) [25]. Lucid is a CNN that takes as input a representation of a traffic flow consisting of packet-level features and returns the probability of the flow being malicious (i.e., part of a DDoS attack). Its input format makes Lucid suitable for data plane feature extraction, where line-rate packet processing is a requirement.

Lucid's representation of traffic flows consists of packet-level features organised in two-dimensional arrays. Rows are the flow's packets in chronological order (Lucid defines bi-directional flows identified by a 5-tuple of IP addresses, L4 ports and L4 protocol), while columns are per-packet features, categorical features included (e.g., TCP flags, IP Flags, ICMP type, etc.). By utilising a convolutional layer as its initial hidden layer, Lucid effectively leverages the aforementioned representation to learn traffic flow semantics and uncover latent behavioural patterns from the chronological sequence of packets. Lucid's output layer is a 1-neuron layer whose value is the probability of the input flow being a DDoS flow.

We set the same hyper-parameters as in the Lucid paper with respect to the height and number of the convolutional kernels, $h = 3$ and $k = 64$ respectively. On the other hand, we slightly adapt the neural network's architecture to comply with the requirements of the feature extraction executed in the data plane. First, two packet features used by Lucid, namely *highest layer* and *protocols*, involve application layer information that is not available in the packet headers (Lucid extracts such features with the support of *TShark* [49], which can return detailed packet's summaries along with standard header fields). Therefore, we only use the $f = 9$ features that can be extracted in the data plane, namely: *Timestamp*, *Packet Length*, *IP Flags*, *TCP Length*, *TCP Ack*, *TCP Flags*, *TCP Window Size*, *UDP Length* and *ICMP Type*.

We also decrease the detection time window from $t = 100$ seconds to $t = 2$ seconds and the number of packets/flow from $p = 100$ to $p = 4$. First, by reducing the input size we reduce the processing time and memory usage. Second, based on the results reported in [25], Lucid can achieve a very high classification accuracy with any value of $t > 1$ second when combined with $p > 3$.

In summary, in our experiments, we use the following Lucid hyper-parameters:

$$\mathbf{p} = 4,\ \mathbf{f} = 9,\ \mathbf{k} = 64,\ \mathbf{h} = 3,\ \mathbf{t} = 2$$

Despite a lower number of packets and features in the flow representations and a shorter time window, we will show that the classification accuracy of the proposed system is comparable to that obtained by the original Lucid implementation on a publicly available dataset of DDoS and benign traffic.

4. System Architecture

Motivated by the insights presented in Section 1, the objective of this study is to enable efficient network intrusion detection in the control plane of programmable networks through packet-level analysis. The primary challenge in achieving this objective lies in the limitations of the network devices responsible for extracting and collecting such features, as well as the constrained capacity of the control channel used to transmit them to the control plane.

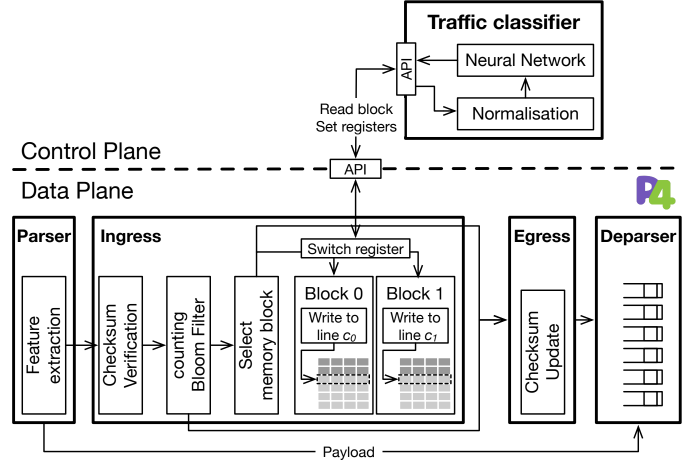


Figure 3: System architecture.

Given these premises, we present P4DDLe, an approach to intrusion detection for P4-based programmable networks that enables resource-efficient traffic feature extraction and filtering in the data plane. The key idea of P4DDLe is to extract and store in the data plane only the features that are essential to the *Traffic Classifier* running on the control plane, thereby optimising the amount of data transmitted through the control channel. For this purpose, we have designed a system architecture matching the *V1model* presented in Section 3.1, with the support of control plane software for traffic classification (Figure 3). The P4 program executed in the data plane extracts the packet attributes and makes them available to the control plane. On the other hand, the control plane is responsible for gathering the traffic metadata from the registers allocated in the data plane and coordinating the read/write access to such registers.

The architecture presented in the figure has been specifically designed to reduce the workload on the control channel while adhering to the limitations imposed by the P4 language and the available resources in the data plane.

4.1. Core logic

Data plane operations are sketched in Figure 3 (bottom part) and elaborated in the pseudo-code of Algorithm 1. The data plane logic is implemented in the *Parser* and in the *Ingress Pipeline*. The *Parser* takes the incoming packets and extracts header fields and metadata required by the *Traffic Classifier* (line 3 of the pseudo-code). The components in the *Ingress Pipeline* are in charge of storing the traffic features and other metadata in stateful memory, provided that the packet passes the *Checksum verification* (line 4).

The key idea behind P4DDLe for supporting packet-level feature extraction and storage is to collect only the features

Algorithm 1 Data plane feature extraction and storage.

```
1:  procedure FeatureExtraction(Packet (pkt))
2:      define p × f  ▷ Traffic Classifier input shape (Sec. 3.3)
3:      hdr, meta ←Parser(pkt)             ▷ Header and metadata
4:      if VerifyChecksum(hdr)==false then
5:          return
6:      end if
7:      f̄ ← meta.features(f)     ▷ Vector of packet's features
8:      id_f^4, id_b^4, id_f^5, id_b^5 ← meta.id    ▷ Set of packet's IDs
9:      for all id ∈ {id_f^4, id_b^4, id_f^5, id_b^5} do
10:         h_id ← CRC32(id)  mod 2^r        ▷ h_id ∈ [0, 2^r − 1]
11:     end for
12:     k ← SwitchReg.Read( )       ▷ Memory index k ∈ {0, 1}
13:     if min_{id∈{id_f^4,id_b^4,id_f^5,id_b^5}} F_k[h_id] < p then
14:         c_k ← PositionReg.Read(k)
15:         Block.Write(B_k[c_k],(id_f^5, f̄))
16:         c_k ← c_k + 1
17:         for all id ∈ {id_f^4, id_b^4, id_f^5, id_b^5} do
18:             if F_k[h_id] < p then
19:                 F_k[h_id]+ = 1         ▷ Increase the counters
20:             end if
21:         end for
22:     end if
23:     ComputeChecksum(h)
24:     DeParser(pkt,hdr)
25: end procedure
```

that are needed by the *Traffic Classifier*. In our implementation, the traffic classifier (i.e., Lucid) takes as input a representation of traffic flows consisting of matrices of packet-level features (Section 3.3). Each flow has the shape $p \times f$, where p denotes the number of packets/flow, and f represents the number of features extracted from the header of each packet. These requirements determine how the packet-level features are extracted and stored in the data plane. Nevertheless, it is important to note that our design is flexible and compatible with various packet-level representations of network traffic, including those involving segments of the packet's payload (e.g., FC-Net [26]).

For each incoming packet, the f features described in Section 3.3 are extracted from its header (line 7). Before storing this information in the stateful memory, P4DDLe verifies that there are less than p packets belonging to the same flow already stored in memory. If this condition is verified, the features are saved in memory, otherwise, they are discarded. By disregarding irrelevant features in the data plane, P4DDLe ensures optimal utilisation of memory resources and control channel.

To achieve this goal, P4DDLe keeps track of the number of packets/flow collected within a given observation time window by using a probabilistic technique based on hashing called *counting Bloom filters* [47] described in Section 3.2. Instead of using h different hash functions to generate h hash values from a packet, P4DDLe leverages the P4 implementation of the CRC32 algorithm with $h = 4$ different packet identifiers $id_f^4, id_b^4, id_f^5, id_b^5$ (line 8). The identifiers id_f^4 and id_b^4 each represent a 4-tuple consisting of source and destination IP addresses, as well as source and destination transport ports, for a given packet. The index f refers to the *forward* order, in which the values appear in the packet, while the index b represents the *backward* order, in which the positions of the IP addresses and transport ports are swapped. The other identifiers id_f^5 and id_b^5 are generated by taking the two 4-tuples id_f^4 and id_b^4, and by adding a fifth item with the value of the packet's transport protocol to each of them (i.e., they both are 5-tuples).

After computing the CRC32 value for each of the four identifiers (as shown in line 10), the algorithm checks whether the minimum counter value among the four counters stored in the filter in the computed positions (i.e., hashed values) $F_k[h_{id}]$ is less than the maximum number of packets/flow allowed (i.e., p). If such a condition is verified, the current packet belongs to a network flow for which less than p packets have been collected so far. In such a case, the packet's 5-tuple identifier id_f^5 and its corresponding features f are saved in memory and the position in the memory is updated. These operations are summarised from line 12 to line 22 of the algorithm's pseudo-code and detailed in the following sections.

Finally, once the feature extraction and collection operations have been completed, the program updates the packet's checksum to reflect any packet modifications (even though the current implementation does not actually alter the packet's contents) (line 23). The header and payload are then reassembled through the DeParser function before the packet is sent to the egress port (line 24).

4.2. Registers

Feature extraction and collection are handled with the support of stateful memory elements, such as registers. The value $k \in \{0, 1\}$ of the *Switch Register* is used to coordinate the memory access and to avoid race conditions between control and data planes. This value is set by the control plane and used by the data plane to select the appropriate registers for read/write operations (line 12).

Traffic features are stored in two memory blocks B_k (line 15). A memory block consists of f registers (one per packet feature), along with other 5 registers for hosting the five elements of the 5-tuple id_f^5, which is used later on by the *Traffic Classifier* to map packets into flows. Each register is divided into n cells, whose size varies depending on the value being stored. In our implementation, register *Source IP* is of size `<bit<32>>`, while *Timestamp* is of size `<bit<48>>`, *TCP Length* is of size `<bit<16>>`, etc. n is the maximum number of packets whose features can be extracted and stored in B_k. Figure 4 sketches the structure of a memory block, in which each row represents a packet, and each column corresponds to an element of the 5-tuple packet identifier or a packet-level feature.

In our implementation, we use two counting Bloom filters F_k, each one associated to the corresponding memory block B_k. Specifically, a filter is a register consisting of m cells, each with a size of 3 bits to store the packets/flow counters. It is important to note that the output of the CRC32 hashing function is a 32-bit number, which would ideally require the allocation of two counting Bloom filters of $m = 2^{32}$ cells. While this would be the

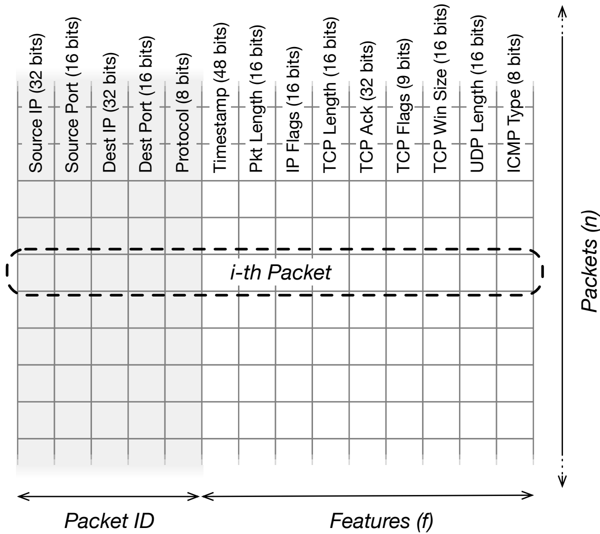


Figure 4: Memory block structure.

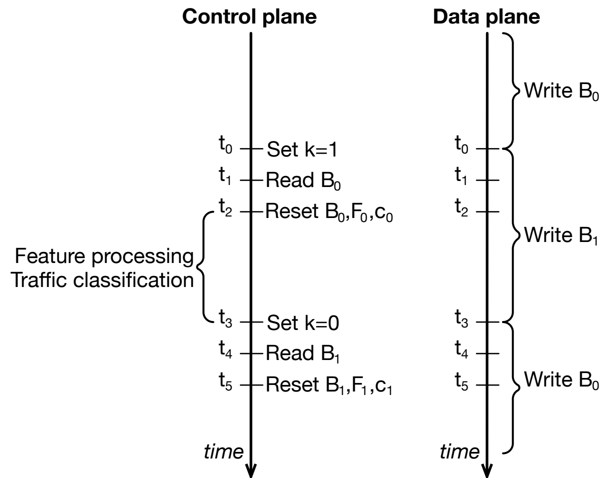


Figure 5: Memory access timeline.

optimal solution to minimize the chances of collisions, such an approach would require a significant amount of memory space. To address this concern, we have decided to limit the bit count of CRC32's output to $r < 32$. We achieve this by using the *modulo* operator (line 10), with r being a number dependent on the number of packets that can be stored in a memory block, denoted by n. The sensitivity of P4DDLe to the size of the counting Bloom Filter is provided in Section 6.1.1.

4.3. Memory Management

The memory blocks B_k are managed as circular buffers to ensure that the most recent traffic data is always available. Thus, when a memory block B_k is full, denoted by $c_k = n-1$, the subsequent packet is stored at the beginning of the block, which corresponds to position $c_k = 0$, overwriting the old packet stored there. When it happens for many packets, such as in high packet rate situations, the control plane classifier may miss the information of older flows either partially or entirely.

4.4. Control-Data Plane interaction

The Control Plane interacts with the Data Plane through Remote Procedure Calls (RPCs), such as those provided by software frameworks like Apache Thrift API [50] or P4Runtime API [51]. In this regard, the Control Plane is in charge of swapping the value of the *Switch Register*, collecting the packet features from the two memory blocks B_k and clearing registers B_k, F_k and c_k ($k \in \{0, 1\}$).

Figure 5 illustrates the timelines of Control and Data Plane operations. The two planes operate in parallel and never block each other. The Data Plane always writes on the same block until the value of the *Switch Register* is changed from the Control Plane. If we start from time t_0, as shown in the figure, once the control plane sets the value of the *Switch Register* to 1, the Data Plane will start writing on memory block B_1. After that, the Control Plane will first read block B_0 and, once done with reading the block, it will reset all the registers of blocks B_0, F_0 and c_0 and the traffic features will be processed for classification. The whole process restarts at time t_3, when the Control Plane sets the *Switch Register* to 0 to retrieve the features collected in memory block B_1, while the Data Plane switches to block B_0. It is important to note that the frequency of this process (computed as $1/(t_3 - t_0)$ using the example in Figure 5) is managed by the Control Plane and does not depend on Data Plane's state.

5. Experimental setup

P4DDLe has been implemented as a set of methods for the reference P4 software switch (namely the Behavioral Model version 2 (BMv2) [52]) and tested in a Mininet emulated network [53]. A prototype implementation of P4DDLe is publicly available for testing and use [54].

The network topology, represented in Figure 6, includes two virtual hosts, one acting as the attacker which sends malicious traffic to the second virtual host (i.e., the victim of the attack). The evaluation environment has been set up on a single physical machine equipped with an 80-core Intel(R) Xeon(R) Gold 5218R CPU @2.10GHz and 128 GB of DDR4 RAM. This machine also runs the Lucid framework [55], which includes a pre-processing tool and a CNN trained for DDoS attack detection.

Lucid and the switch are interfaced through Remote Procedure Calls (RPCs). The RPC mechanism is implemented using the Apache Thrift API [50] and is used to read, write and reset the registers from the control plane through data plane methods `bm_register_read`, `bm_register_write` and `bm_register_reset` respectively [56].

5.1. Dataset

P4DDLe is evaluated using CIC-DDoS2019 [57], a recent dataset of DDoS attacks provided by the Canadian Institute of Cybersecurity at the University of New Brunswick. This dataset contains multiple days of network activity, including benign

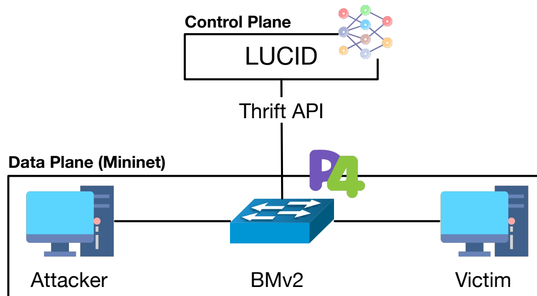


Figure 6: Experimental setup.

traffic and 13 distinct types of DDoS attacks. It is accessible to the public and contains pre-recorded traffic traces with complete packet payloads, along with supplementary text files that provide labels and statistical information for each traffic flow [58]. In our experiments, we inject the P4 switch with the traffic traces to evaluate P4DDLe's performance in handling the feature extraction process in the data plane. In the dataset, the benign traffic was created using the B-profile [59], which defines distribution models for various applications, such as web (HTTP/S), remote shell (SSH), file transfer (FTP), and email (SMTP). On the other hand, the attack traffic was generated using third-party tools and can be broadly classified into two main categories: *reflection-based* and *exploitation-based* attacks. The first category involves attacks in which the attacker triggers responses from a remote server (such as a DNS resolver) towards the victim's spoofed IP address, ultimately overwhelming the victim with these responses. The second category pertains to attacks that exploit known vulnerabilities in target systems, applications or in certain network protocols.

The traffic traces have been divided into training, validation and test sets and processed with the Lucid's packet parser [60]. Such a tool extracts packet-level features from the traffic and groups them into array-like representations of network flows, as described in Section 3.3.

Table 1: DDoS attack traces (test sets).

Attack	Duration (sec)	#Flows	Flow rate (flows/sec)	Packet rate (kpackets/sec)
DNS	383	5736	15	2.6
LDAP	27	26	0.96	8
MSSQL	65	787092	12109	24
NetBIOS	65	519069	7985	16
Portmap	19	152956	8050	16
SNMP	38	74882	1970	17
SSDP	23	195521	8500	30
UDP-Lag	13	116439	8957	20

The pre-processing of the training and validation traces is done offline for Lucid's training and tuning, whereas the traffic traces of the test set are solely employed for online testing. During the testing phase, feature extraction is performed in the data plane (as explained in Section 4), while Lucid's parser tool is executed in the control plane for building the arrays and normalising the features.

To mitigate the impact of BMv2's poor performance [61], we made a strategic decision to exclude all attacks with a packet rate higher than 30 kpackets/s from the test set. This step was necessary to ensure the integrity and accuracy of our evaluation results by preventing any interference caused by packets being dropped due to BMv2's performance limitations. The key details of the remaining test traces are presented in Table 1 for reference.

5.2. Evaluation metrics

Our primary evaluation metrics are the *Collected Flows* and the *system False Negative Rate (sFNR)*. *Collected Flows* measures the number of flows stored in memory relative to the total number of flows injected into the switch within a given time window. This metric provides insight into P4DDLe's ability to capture information on as many traffic flows as possible.

The sFNR quantifies the percentage of malicious flows that go undetected, either due to misclassification by the NIDS, or because no packets of such flows are present in the memory block due to the reasons explained in Section 4.3. By assessing the sFNR, we measure the efficiency of P4DDLe in promptly identifying and flagging potential intrusions. A formal definition of the sFNR measured in a given observation time window is provided in Equation 1.

$$sFNR = FNR + \frac{1}{|X_m|} \sum_{x_i \in X_m} c_{x_i} \oplus n_{x_i} \tag{1}$$

In Equation 1, FNR is the False Negative Rate of the NIDS running in the control plane. It is worth reminding that FNR is the metric that measures the percentage of positive samples (in our case, the malicious flows) that are misclassified as negatives by a classifier. In the equation, the set of malicious flows is denoted by X_m, c_{x_i} represents the actual number of packets collected for a given malicious flow x_i in a memory block, while n_{x_i} is the number of observed packets of that flow. The *XOR* operation $c_{x_i} \oplus n_{x_i}$ returns 1 if $c_{x_i} = 0$ and $n_{x_i} \neq 0$ or if $c_{x_i} \neq 0$ and $n_{x_i} = 0$ (the second case can never happen), and returns 0 otherwise. The summation computes the total number of observed malicious flows in a given time interval with no packets collected in the memory block. We divide this value by the total number of observed malicious flows X_m to obtain the rate.

We also define the *quality* metric that allows us to establish a relationship between three distinct quantities: (i) the number of packets in a traffic flow, (ii) the number of packets of that flow collected in the data plane within a given time window, and (iii) the packet/flow p required by the *Traffic Classifier* running in the control plane. This metric quantifies the level of "useful" information that the data plane delivers to the Traffic Classifier for each flow, while taking into account that gathering more than p packets per flow is inefficient resource utilization. Specifically, the quality metric is defined by Equation 2.

$$quality = \frac{1}{|X|} \sum_{x_i \in X} \frac{c_{x_i}}{l_{x_i}} \tag{2}$$

The quality metric is determined by taking the average quality score across the flows observed within a given time window. In the equation, the set of such flows is denoted by $X = \{X_b, X_m\}$ and includes both benign flows X_b and malicious flows X_m. The quality of a single flow $x_i \in X$ is expressed as c_{x_i} over l_{x_i}, where $l_{x_i} = \min\{n_{x_i}, p\}$ is the minimum between the number of packets n_{x_i} of the i-th flow and p. l_{x_i} represents the optimal number of packets that we need to collect in the data plane to maximise the amount of information provided to the ML model for a given flow x_i. Collecting fewer packets may not provide sufficient information for the classifier, while collecting more may waste valuable memory space without providing any additional benefits to the classifier. This is because any excess packets beyond the optimal number would be discarded by the feature pre-processing component, which constructs the flow samples required by the ML model.

In the case of the quality metric, c_{x_i} represents the number of packets stored in a memory block for a given flow x_i, either benign or malicious. With P4DDLe, which uses a counting Bloom filter to track the number of packets/flow in memory, the value of c_{x_i} ranges from 0 to p. With no packet tracking, c_{x_i} may fall between 0 and n_{x_i}. It is worth noting that in certain cases, the value of c_{x_i} may be zero even when $n_{x_i} \neq 0$. This can occur when no packets for flow x_i have been collected in the current memory block, either because of collisions or because they were overwritten by more recent packets in the circular buffer (Section 4.3).

6. Evaluation Results

One of the key benefits of P4DDLe over other approaches is the ability to extract raw packet features from the network traffic, categorical features included, and to organise them in a way that the semantics of traffic flows are preserved. P4DDLe efficiently achieves this objective by implementing a counting Bloom filter (Section 3.2) that tracks the number of packets/flow without any wastage of memory resources in the data plane. We demonstrate the advantages of P4DDLe through simulation and emulation experiments, where we disable the tracking methods and observe the corresponding changes using a range of metrics.

In the simulation scenario, no actual network data is involved. Instead, we rely on the network profile of a campus network [62] and we demonstrate the benefits of P4DDLe by simulating the feature extraction and storage process in the data plane. The second set of experiments has been conducted in an emulated environment consisting of two hosts (an attacker and a victim) and a P4-enabled software switch with configurable registers. The experiments involve injecting pre-recorded network traffic into this environment, using a publicly available dataset of both benign and DDoS traffic.

6.1. Simulation Scenario

The goal of the simulations is to analyse the correlation between the size of the memory block B_k and two key variables: the maximum number of packets/flow (i.e., variable p) and the number of cells m of the Bloom filter F_k. This information gives an understanding on how to configure P4DDLe based on the characteristics of the network traffic under monitoring (mainly the average packet rate and average packets/flow) to maximise the number of collected flows within a given time window. To do so, we fix the size of B_k and we vary the value of p and the bit count r of the CRC32's output that determines the size of the Bloom filter (see Section 4.2).

We utilise the profile outlined in [62] to simulate the benign traffic of a realistic network. This profile was generated using network activity data gathered from a university campus and does not present any documented attack in the trace. From it, we extracted the distribution of TCP, UDP, and other protocols and we computed the average number of packets/flow for each protocol.

Each experiment consists of 1000 iterations, each simulating the collection of a random number of flows, ranging from 1 to 8192 flows. The length of such flows (number of packets/flow) is generated based on the aforementioned traffic profiles, while the arrival of packets across different flows has been randomised to ensure a non-sequential packet distribution, aligning with the characteristics of realistic packet-switched networks. To minimise potential issues due to the insufficient space in memory to store packets for all the 8192 flows (which is not the goal of this analysis), we reserve space in the memory block B_k for at least two packets/flow by setting the value $n = 16384$ packets (see also Figure 4 for reference).

These experiments are realised with Python scripts designed to simulate a single collection of network flows. The scripts replicate the logic of both P4DDLe (as described in Algorithm 1) and a baseline configuration. In the baseline setup, the packet filtering algorithm intrinsic to P4DDLe is deliberately deactivated, causing all incoming packets to be stored in memory within the circular buffer, with no constraints on the number of packets per flow. These scripts are publicly available for testing on the P4DDLe repository [54].

6.1.1. Bloom filter size

Each Bloom filter is characterised by two key dimensions: the number of cells, denoted as $m \in [0, 2^r - 1]$, and the size of each cell. In this experiment, we have set the size of the cells while varying the value of r to understand the sensitivity of P4DDLe to the number of cells in terms of *Collected Flows*. Considering that 75% of the flows within the traffic profile consist of no more than 4 packets, we have chosen to set the cell size to 3 bits. This size adequately accommodates the counting of up to 4 packets/flow.

Figure 7 presents the results obtained with Bloom filter sizes from 1024 to 32768 cells (or r from 10 to 15). The figure also includes the results obtained with a single hash function (with $r = 15$) , allowing for a comparative analysis with Bloom filters. As expected, the larger the Bloom filter size, the smaller the number of hash collisions and the higher the number of flows collected. Remarkably, it is worth noting that P4DDLe is able to collect more flows than the baseline, regardless of the Bloom filter size. It is important to recall that the baseline approach gathers all packets without any filtering logic. Considering that

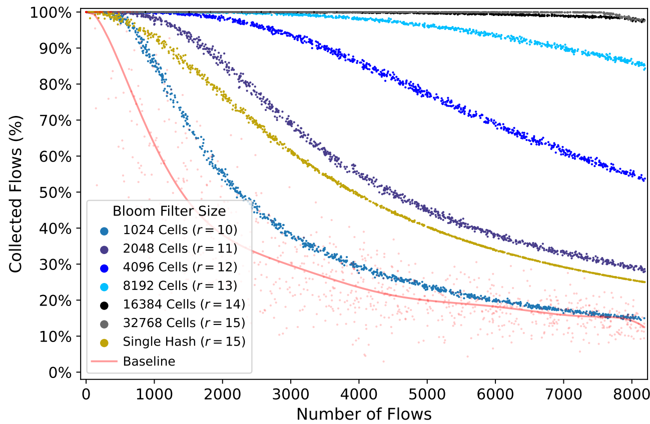


Figure 7: Percentage of collected flows on the value of r.

the average number of packets/flow in the traffic profile is 78, the baseline approach exhausts the memory capacity with approximately 210 flows (calculated as 16384/78). Consequently, once the memory block reaches its limit, older packets are overwritten by more recent ones, resulting in a loss of flows. The results of this experiment also confirm the efficiency of Bloom filters over a single hash function in limiting the collisions, as discussed in Section 3.2 and demonstrated by Tarkoma et al. [47].

Figure 7 leads to another significant observation: the stability of P4DDLe throughout the experiment rounds, in contrast to the baseline approach. This stability is a direct result of the packet filtering strategy employed by P4DDLe. By filtering out excess packets from each flow, P4DDLe ensures that the memory allocated for a flow remains bounded by the predetermined maximum p. Consequently, the number of collected flows remains stable even in extremely randomised scenarios like this simulation. In contrast, the baseline approach lacks any form of packet filtering. As a result, the number of packets per flow stored in memory is not constrained, leading to much larger fluctuations in terms of collected flows. This absence of bounds on per-flow packet storage is evident from the erratic behaviour depicted in the figure, where the scattered values of collected flows reflect the wide range of flow sizes, spanning from 1 to 1000 packets (with average and standard deviation of 77 and 74 packets, respectively). The baseline curve shown in the figure represents a polynomial approximation of the values and serves to emphasize the average trend of collected flows under this configuration.

In light of the results obtained in this simulation, we have determined that setting the number of cells to 32768 is the most suitable choice. This value effectively minimises hash collisions (0.5% with 8000 flows) while maintaining a minimal impact on memory usage. It is important to note that P4DDLe utilises two counting Bloom filters (F_k with $k \in \{0, 1\}$ in Algorithm 1). Since we only need 3 bits to count up to $p = 4$ packets/flow, the size of each F_k Bloom filter will be 12 kilobytes. This memory requirement is relatively insignificant when compared to the memory blocks B_k, each of which demands approximately 562 kilobytes to store the features of 16384 packets.

6.1.2. *Maximum number of collected packets per flow*

In this experiment, we fix the size of the Bloom filter to 32768 cells, and we vary p from 2 to 128 packets/flow. We expect that by decreasing the value of p, the average number of packets collected per flow will also decrease, leading to a higher number of collected flows in memory but also to more potential collisions.

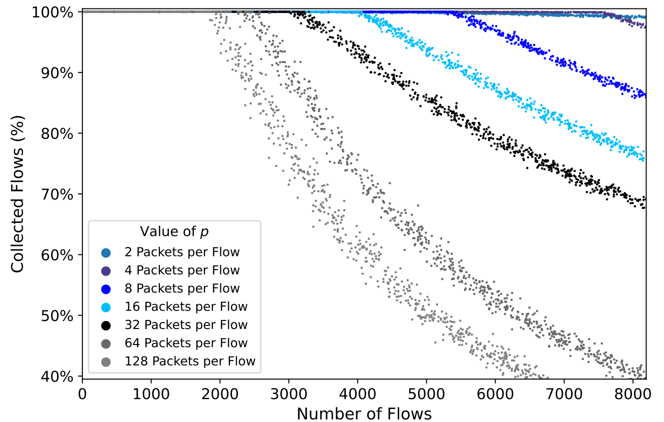


Figure 8: Percentage of collected flows on the value of p.

This behaviour can be observed in Figure 8, which demonstrates that by increasing p, the memory block fills up sooner and, as a consequence, the percentage of collected flows deteriorates earlier. Of course, with $p = 2$ the memory is never filled, even with 8192 generated flows (we remind that we set the size of the memory block to 16384 packets). However, we can notice that with $p = 2$ a small portion of flows is lost due to collisions when the number of generated flows is 5000 or higher. Indeed, when the value of p is small, there is an increased likelihood of missing the condition at line 13 of Algorithm 1. As a result, a higher number of packets and flows are discarded. In the figure, we can also notice the stability of P4DDLe with $p = 4$ almost until the end of the experiment. For this reason, and considering the proven effectiveness of LUCID with $p > 3$ (as discussed in Section 3.3), we set $p = 4$ for the experiments conducted in the emulated environment.

6.2. *Emulated Scenario*

Based on the results of the simulations described in the previous section, we set the parameters for the emulations as follows: (i) memory block size: $n = 16384$ packets, (ii) bloom filter size: 32768 cells ($r = 15$) and (iii) maximum packets/flows $p = 4$. The duration of all the experiments has been set to 6 minutes, during which we inject various combinations of attack and benign traffic from the attacking host to the victim through the P4DDLe-enabled switch (see Figure 6).

In this experiment, we generate a series of DDoS attacks by using the traffic traces of the CIC-DDoS2019 dataset (see

Section 5.1). If we exclude the DNS attack, the flows of the other attacks present an average flow length of around 2.2 packets, hence considerably smaller than those of the profile of the campus network used in simulations (around 78 packets/flow). Considering that we chose $p = 4$ (i.e., $p > 2.2$), we expect that with these small flows, both P4DDLe and the baseline can collect approximately the same number of flows. Acknowledging that malicious traffic flows typically do not constitute the totality of network traffic, we define four evaluation scenarios to enable a comprehensive comparison. These scenarios involve the inclusion of background benign traffic at varying proportions relative to the total packets in the network, namely 0% (no benign traffic), 25%, 50%, and 75%. Table 2 presents the average flow length as we increase the percentage of benign traffic in the network (whose average packet rate is about 0.7 packets/sec and average flow length is about 36 packets).

Table 2: Impact of benign traffic on the average flow length.

Attack	**Average Flow Length**			
	0%	**25%**	**50%**	**75%**
SSDP	3.5	4.4	6.0	9.7
MSSQL	2.0	2.7	3.8	6.8
UDP-Lag	2.2	2.9	4.1	7.3
SNMP	2.0	2.6	3.8	6.7
Portmap	2.0	2.6	3.7	6.6
NetBIOS	2.0	2.6	3.7	6.6
LDAP	2.0	2.6	3.7	6.6
DNS	109.3	76.3	66.0	49.8
Average	**15.6**	**12.1**	**11.9**	**12.5**
Average (w/o DNS)	**2.2**	**2.9**	**4.1**	**7.2**

Other important information pertains to the relationship between packets and flows. Since we add a portion of background benign traffic based on the total amount of packets, this does not necessarily reflect an analogous increase in the number of flows. Actually, based on the low average flow length in the attacks (first column in Table 2), and the relatively high number of packets/flow of the benign trace (around 36 on average), even splitting 50% between benign and DDoS packets, the average percentage of DDoS flows is higher than 93% on the total of flows (excluding the DNS attack that presents an exceptional behaviour). Table 3 shows the percentage of DDoS flows for each attack type.

Given these premises, we compare P4DDLe against the baseline approach in the emulated environment. For both approaches, we inject each of the eight attacks of the CIC-DDoS2019 dataset combined with benign traffic in the various proportions reported above. The comparison is evaluated using the three metrics presented in Section 5.2, namely *sFNR*, *Collected Flows* and *quality*.

6.2.1. Collected Flows and sFNR

Figure 9 shows the comparison related to *Collected Flows* for all attack types. First, we can notice that with the baseline approach the percentage of collected flows remains stable when

Table 3: Percentage of DDoS Flows based on the volume of benign packets.

Attack	**Percentage of DDoS Flows**			
	0%	**25%**	**50%**	**75%**
SSDP	100%	96.01%	90.07%	76.78%
MSSQL	100%	97.11%	93.00%	83.40%
UDP-Lag	100%	97.40%	93.19%	82.50%
SNMP	100%	97.63%	93.72%	83.45%
Portmap	100%	97.70%	93.57%	83.16%
NetBIOS	100%	97.71%	93.51%	83.20%
LDAP	100%	96.99%	93.48%	83.16%
DNS	100%	46.89%	27.15%	14.87%
Average	**100%**	**90.93%**	**84.71%**	**73.82%**

we vary the proportion between benign and attack traffic. This can be attributed to the absence of any filtering logic, causing the percentage of collected flows to solely depend on the memory capacity. In contrast, P4DDLe demonstrates consistent improvement in the percentage of collected flows as the volume of benign traffic (and the average flow length) increases. While the baseline approach fills up the memory block predominantly with large benign flows, P4DDLe optimizes memory usage by storing only the essential information required by the NIDS and filtering out packets that exceed the maximum limit of $p = 4$. This memory management prevents large flows from occupying excessive memory space, thus ensuring that a higher number of flows can be transmitted to the NIDS for classification. For instance, in the case of the MSSQL attack, whose average flow size is around 2 packets/flow, with 0% of benign traffic the performance of P4DDLe is approximately the same as with the baseline approach. This is because with $p = 4$, no packets are filtered and the memory occupation is similar with both approaches. However, as soon as we add some benign traffic, which is composed of larger flows, the filtering mechanism starts dropping the extra packets, saving space in memory for more flows.

P4DDLe inherently maintains a balanced allocation of memory for both low-rate and high-rate flows, resulting in similar probabilities for their storage. This balance can be observed when analyzing the distribution of collected flows, distinguishing between DDoS traffic, whose packet rate ranges from 2.6K to 30K packets per second, and benign flows, which average around 0.7 packets per second. For instance, consider the SSDP attack scenario with 30K packets/sec, where the test traffic is divided into 25% benign and 75% attack traffic. In this worst-case scenario, the average percentage of collected flows is approximately 42% for benign flows and 46% for attack flows, demonstrating P4DDLe's stability and effectiveness in handling heterogeneous network conditions.

The sFNR metric (Figure 10) shows the ability of P4DDLe to collect more malicious flows than the baseline approach, ultimately leading to faster mitigation of network attacks. It is also worth noticing that when the volume of the attack is small, like in the case of LDAP and DNS, the memory is never filled, even when adding benign traffic. Therefore, the collected flows

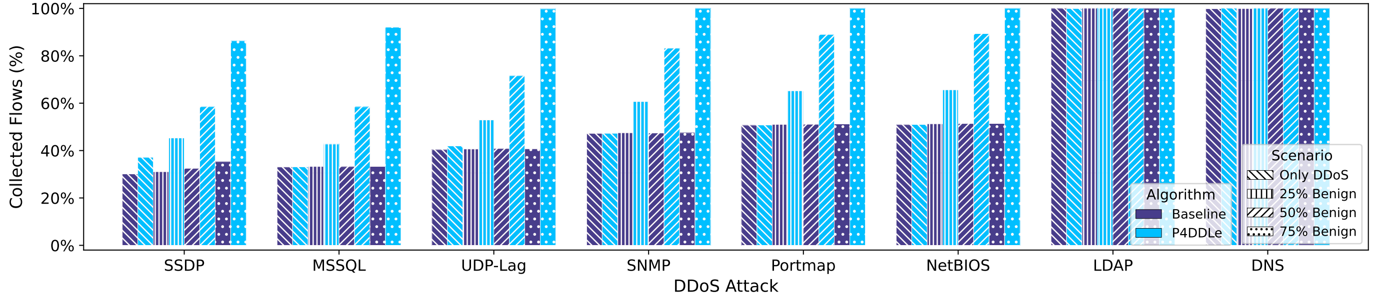


Figure 9: Average Collected Flows.

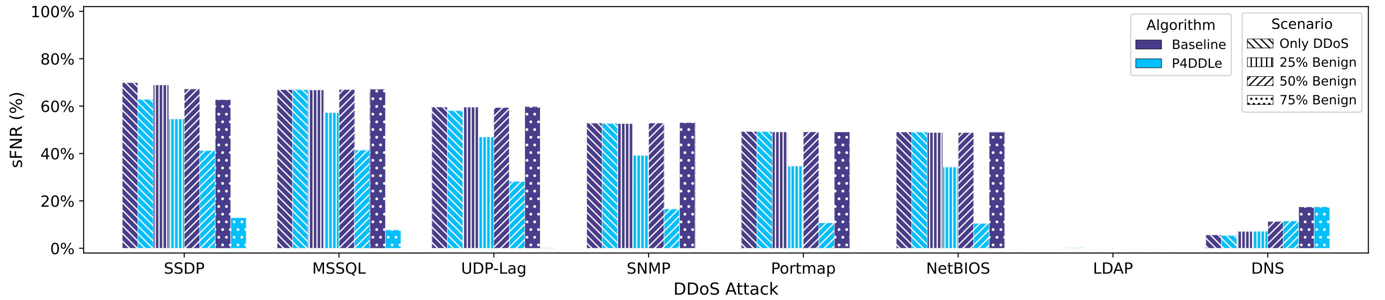


Figure 10: Average sFNR.

is 100% for both, while the non-zero sFNR obtained with the DNS attacks is solely due to Lucid's FNR.

6.2.2. *Flow quality*

As defined in Section 5.2, the quality metric measures the average amount of "useful" data collected in the data plane and transmitted to the control plane for classification. Table 4 presents the average quality metrics, revealing that while P4DDLe can collect a larger number of flows in memory, these flows maintain a high level of quality, although sometimes slightly lower than that obtained with the baseline.

Unlike the baseline, P4DDLe might drop packets because of collisions, which accounts for the slight reduction in quality observed in the attack scenarios that present a combination of high flow rate and high packet rate. The SSDP attack is an interesting use case that demonstrates this observation: as we add more and more benign traffic, the packet rate remains constant (around 30 kpackets/sec), while the flow rate decreases, hence the chances of collisions.

6.2.3. *Temporal evolution*

A comparison between the two approaches throughout the whole experiment is presented in Figures 11 and 12. For a matter of space, we provide the plots of only two attacks, although the other attacks present similar behaviours. The plotted data consistently validates the average numbers presented earlier in this section, with no notable deviations observed over time.

Besides the memory size and the packet rate, the performance of P4DDLe is also influenced by the flow size and, as a consequence, by the flow rate. By collecting only $p = 4$ packets/flow, the number of collected flows increases (and the sFNR decreases) when the flow rate increases, as highlighted by the respective plots in the two Figures. On the contrary, the baseline approach is less affected by the flow rate, as it keeps on overwriting the old flows when the memory is full, resulting in approximately a stable sFNR and number of collected flows.

Table 4: Quality of collected flows at various percentages of benign packets.

Attack	Baseline				P4DDLe			
	0%	**25%**	**50%**	**75%**	**0%**	**25%**	**50%**	**75%**
SSDP	95.7	94.8	93.3	86.2	90.1	91.9	93.8	95.7
MSSQL	100.0	100.0	100.0	100.0	99.9	99.5	99.3	99.7
UDP-Lag	99.7	99.8	99.9	99.9	97.8	98.4	99.2	99.8
SNMP	100.0	100.0	100.0	100.0	100.0	99.8	99.8	99.9
Portmap	100.0	100.0	100.0	100.0	100.0	99.8	99.8	99.9
NetBIOS	100.0	100.0	100.0	100.0	100.0	99.8	99.8	99.9
LDAP	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
DNS	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Average	**99.4**	**99.3**	**99.1**	**98.3**	**98.5**	**98.6**	**99.0**	**99.4**

6.3. *Control Plane performance*

As described in Section 3.3, the validation of P4DDLe has been carried out using a modified configuration of Lucid, the ML-based NIDS operating in the control plane. This adaptation is essential to overcome the constraints posed by the P4 data plane. Notably, two key features are omitted (*highest layer* and *protocols*), and the packet count per flow p is significantly

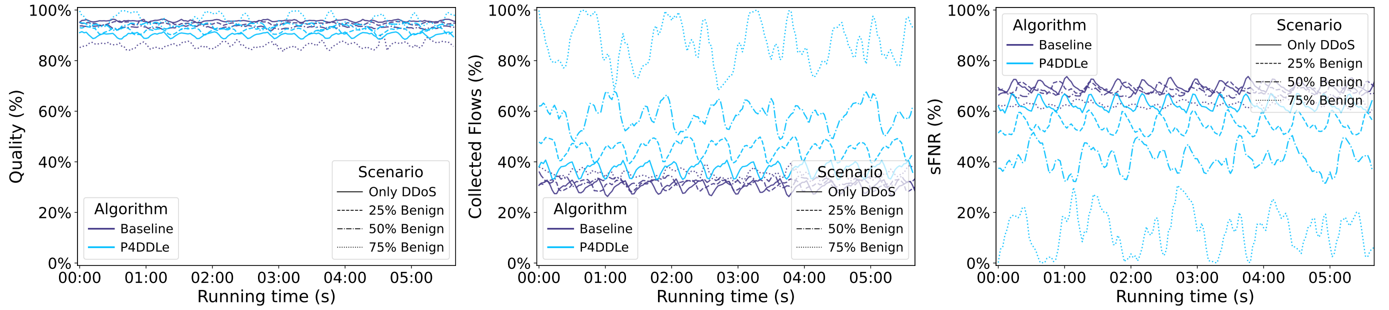


Figure 11: Metrics by running time on SSDP DDoS Attack.

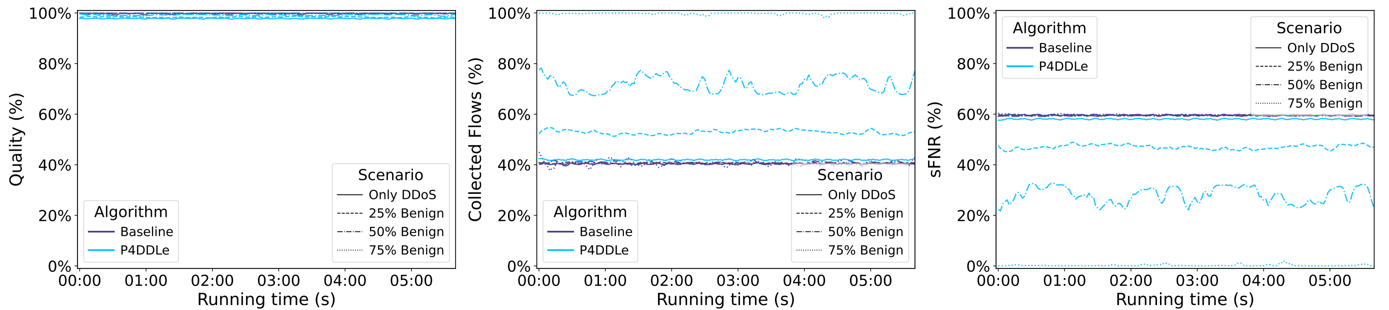


Figure 12: Metrics by running time on UDP-Lag DDoS Attack.

reduced from 100 to 4. Under these settings, we measured an average FNR of approximately 0.013 and an average accuracy and F1 Score of about 0.93 on the CIC-DDoS2019 dataset across the experiments described in Section 6.2.

Nevertheless, we are keen to assess the impact of our adaptation on the Lucid's overall performance. To achieve this objective, we have conducted further experiments in the emulated environment, replicating the three DDoS attacks used in the Lucid's original paper. The three attacks, generated using an IRC botnet and with well-known DDoS attacking tools such as LOIC [63] and HOIC [64], are part of three datasets provided by the University of New Brunswick, namely ISCX2012 [65], CIC2017 [66] and CSECIC2018 [67].

The results presented in Table 5 showcase the performance assessment of Lucid in segregating DDoS attacks from legitimate traffic under *offline* and *online* settings. The offline results are derived from Lucid's original paper, where the evaluation was based on a static dataset of traffic flows represented by 11 features, including *highest layer* and *protocols*, and with $p = 100$. Conversely, the online configuration aligns with the experimental setup detailed in earlier sections, featuring 9 features and $p = 4$.

As reported in the table, the impact of reducing the feature matrix size from 100×11 to 4×9 is relatively insignificant in the case of the HOIC-based DDoS attack, with a noticeable but moderate increase in the FPR from 0.16% to 1.5%. However, for the other attacks, we can observe a decrease of approximately 3% in the F1 Score, indicating a general decline in overall performance, including a higher FPR in both instances.

It is worth mentioning that the *online* results were obtained by training Lucid with just 9 features and 4 packets per flow. This highlights once again the remarkable adaptability of neural networks in adjusting their weights to the available input, underlining their flexibility and robustness in various configurations.

7. Conclusion

In this paper, we have presented P4DDLe, a solution based on P4 programmable data planes that enables the benefits of centralised intrusion detection while reducing the impact on control channel and hardware resources. P4DDLe takes advantage of a probabilistic hashing data structure to carefully select the amount of information to be extracted from the network traffic and sent to the control plane, taking into consideration the traffic features required by the traffic classifier.

The key advantage of P4DDLe over similar solutions resides in its ability to build a packet-level representation of traffic flows. This peculiarity allows P4DDLe to satisfy the requirements of state-of-the-art ML-based NIDS, which rely on a detailed representation of the traffic that goes beyond mere statistics. We have demonstrated that by using a counting Bloom filter to retain only the necessary information within the switch, P4DDLe optimises the usage of stateful memory in the data plane, while reducing the chances of missed malicious flows due to lack of memory space.

Table 5: Control plane performance

Dataset	Accuracy		False Positive Rate		Precision		Recall		F1 Score	
	Offline	Online	Offline	Online	Offline	Online	Offline	Online	Offline	Online
ISCX2012 (IRC)	0.9888	0.9612	0.0179	0.0612	0.9827	0.9313	0.9952	0.9874	0.9889	0.9584
CIC2017 (LOIC)	0.9967	0.9455	0.0059	0.0123	0.9939	0.9994	0.9994	0.9430	0.9966	0.9697
CSECIC2018 (HOIC)	0.9987	0.9922	0.0016	0.0154	0.9984	0.9978	0.9989	0.9978	0.9987	0.9956

References

[1] European Union Agency for Cybersecurity (ENISA), “ENISA Threat Landscape 2022,” https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022, 2022, [Accessed: 30-June-2023].

[2] E. Rojas, R. Doriguzzi-Corin, S. Tamurejo, A. Beato, A. Schwabe, K. Phemius, and C. Guerrero, “Are we ready to drive software-defined networks? a comprehensive survey on management tools and techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–35, 2018.

[3] S. Khorsandroo, A. G. Sánchez, A. S. Tosun, J. M. Arco, and R. Doriguzzi-Corin, “Hybrid sdn evolution: A comprehensive survey of the state-of-the-art,” *Computer Networks*, vol. 192, p. 107981, 2021.

[4] M. E. Kanakis, R. Khalili, and L. Wang, “Machine learning for computer systems and networking: A survey,” *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–36, 2022.

[5] N. Sultana, N. Chilamkurti, W. Peng, and R. Alhadad, “Survey on sdn based network intrusion detection system using machine learning approaches,” *Peer-to-Peer Networking and Applications*, vol. 12, pp. 493–501, 2019.

[6] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, “Dshark: A general, easy to program and scalable framework for analyzing in-network packet traces,” ser. NSDI’19. USENIX Association, 2019.

[7] S. Wang, C. Sun, Z. Meng, M. Wang, J. Cao, M. Xu, J. Bi, Q. Huang, M. Moshref, T. Yang, H. Hu, and G. Zhang, “Martini: Bridging the gap between network measurement and control using switching asics,” in *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, 2020.

[8] X. Chen, H. Liu, D. Zhang, Q. Huang, H. Zhou, C. Wu, and Q. Yang, “Eliminating control plane overload via measurement task placement,” *IEEE/ACM Transactions on Networking*, pp. 1–15, 2022.

[9] H. Liu, X. Chen, Q. Huang, D. Kong, J. Sun, D. Zhang, H. Zhou, and C. Wu, “Escala: Timely elastic scaling of control channels in network measurement,” in *IEEE INFOCOM 2022*, 2022.

[10] C. H. Song, X. Z. Khooi, D. M. Divakaran, and M. C. Chan, “Revisiting application offloads on programmable switches,” in *2022 IFIP Networking Conference (IFIP Networking)*, 2022.

[11] T. Bühler, R. Jacob, I. Poese, and L. Vanbever, “Enhancing global network monitoring with magnifier,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1521–1539.

[12] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, “The programmable data plane: Abstractions, architectures, algorithms, and applications,” *ACM Comput. Surv.*, vol. 54, no. 4, may 2021.

[13] J. Xing, Q. Kang, and A. Chen, “Netwarden: Mitigating network covert channels while preserving performance,” in *USENIX Security*, 2020.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014.

[15] D. Ding, M. Savi, and D. Siracusa, “Estimating logarithmic and exponential functions to track network traffic entropy in p4,” in *Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020.

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

[17] F. Musumeci, A. C. Fidanci, F. Paolucci, F. Cugini, and M. Tornatore, “Machine-learning-enabled ddos attacks detection in p4 programmable networks,” *Journal of Network and Systems Management*, vol. 30, no. 1, pp. 1–27, 2022.

[18] S. G. Macías, L. P. Gaspary, and J. F. Botero, “Oracle: An architecture for collaboration of data and control planes to detect ddos attacks,” in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 962–967.

[19] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. Ramos, and A. Madeira, “Flowlens: Enabling efficient flow classification for ml-based network security applications.” in *NDSS*, 2021.

[20] M. Zang, E. O. Zaballa, and L. Dittmann, “Sdn-based in-band ddos detection using ensemble learning algorithm on iot edge,” in *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. IEEE, 2022, pp. 111–115.

[21] M. Roshani and M. Nobakht, “Hybriddad: Detecting ddos flooding attack using machine learning with programmable switches,” in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–11.

[22] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, “Ssh compromise detection using netflow/ipfix,” *ACM SIGCOMM computer communication review*, vol. 44, no. 5, pp. 20–26, 2014.

[23] A. Custura, R. Secchi, and G. Fairhurst, “Exploring dscp modification pathologies in the internet,” *Computer Communications*, vol. 127, pp. 86–94, 2018.

[24] P. Illy, G. Kaddoum, K. Kaur, and S. Garg, “Ml-based idps enhancement with complementary features for home iot networks,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 772–783, 2022.

[25] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del Rincón, and D. Siracusa, “Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 876–889, 2020.

[26] C. Xu, J. Shen, and X. Du, “A method of few-shot network intrusion detection based on meta-learning framework,” *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3540–3552, 2020.

[27] M. M. Alani, “Botstop: Packet-based efficient and explainable iot botnet detection using machine learning,” *Computer Communications*, vol. 193, pp. 53–62, 2022.

[28] E. Min, J. Long, Q. Liu, J. Cui, , and W. Chen, “TR-IDS: Anomaly-Based Intrusion Detection through Text-Convolutional Neural Network and Random Forest,” *Security and Communication Networks*, 2018.

[29] X. Han, S. Cui, S. Liu, C. Zhang, B. Jiang, and Z. Lu, “Network intrusion detection based on n-gram frequency and time-aware transformer,” *Computers & Security*, vol. 128, p. 103171, 2023.

[30] B. Coelho and A. Schaeffer-Filho, “Backorders: Using random forests to detect ddos attacks in programmable data planes,” in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, p. 1–7.

[31] Q. Qin, K. Poularakis, K. K. Leung, and L. Tassiulas, “Line-speed and scalable intrusion detection at the network edge via federated learning,” in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 352–360.

[32] D. Ding, M. Savi, and D. Siracusa, “Tracking normalized network traffic entropy to detect ddos attacks in p4,” *IEEE Transactions on Dependable and Secure Computing*, 2021.

[33] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor *et al.*, “Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation,” in *International Conference on High Performance Computing*. Springer, 2020, pp. 41–59.

[34] Y. Yuan, O. Alama, J. Fei, J. Nelson, D. R. Ports, A. Sapio, M. Canini, and N. S. Kim, “Unlocking the power of inline {Floating-Point} operations on programmable switches,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 683–700.

[35] X. Zhang, L. Cui, F. P. Tso, and W. Jia, “pheavy: Predicting heavy flows

in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[36] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Map4: A pragmatic framework for in-network machine learning traffic classification," *IEEE Transactions on Network and Service Management*, 2022.

[37] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Iisy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.

[38] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[39] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation," in *IEEE INFOCOM 2022*, 2022.

[40] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pforest: In-network inference with random forests," *arXiv preprint arXiv:1909.05680*, 2019.

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

[42] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, G. Antichi, P. Costa, H. Haddadi, and R. Bifulco, "Re-architecting traffic analysis with neural network interface cards," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[43] K. Razavi, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "Distributed dnn serving in the network data plane," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, p. 67–70.

[44] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Automating in-network machine learning," *arXiv preprint arXiv:2205.08824*, 2022.

[45] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks," *arXiv preprint arXiv:2206.05592*, 2022.

[46] C. B. Serna and C. Mas-Machuca, "Preventing control plane overload in sdn networks with programmable data planes," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022.

[47] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[48] P. Manzanares-Lopez, J. P. Muñoz-Gea, and J. Malgosa-Sanahuja, "Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry," *IEEE Access*, vol. 9, pp. 20 391–20 409, 2021.

[49] G. Combs, "Tshark - dump and analyze network traffic," 2022, [Accessed: 30-Nov-2022]. [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html

[50] Apache Software Foundation, "Apache Thrift," 2022, [Accessed: 30-Jun-2023]. [Online]. Available: https://thrift.apache.org/

[51] The P4.org API Working Group, "P4Runtime Specification," 2022, [Accessed: 30-Jun-2023]. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[52] Open Networking Foundation, "Behavioral Model Source Code," 2022, [Accessed: 30-Jun-2023]. [Online]. Available: https://github.com/p4lang/behavioral-model

[53] Mininet Project Contributors, "Mininet," 2022, [Accessed: 30-Jun-2023]. [Online]. Available: http://mininet.org/

[54] Luis Augusto Dias Knob, "P4DDLe source code," 2023, [Accessed: 30-Jun-2023]. [Online]. Available: https://github.com/risingfbk/p4ddle

[55] R. Doriguzzi-Corin, "LUCID source code," https://github.com/doriguzzi/lucid-ddos, 2020, [Accessed: January 16, 2024].

[56] Antonin Bas, "Behavioral Model Thrift Source Code," 2022, [Accessed: 30-Jun-2023]. [Online]. Available: https://github.com/p4lang/behavioral-model/tree/main/thrift_src

[57] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (ddos) attack dataset and taxonomy," in *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2019, pp. 1–8.

[58] University of New Brunswick. (2019) DDoS Evaluation Dataset. [Online]. Available: https://www.unb.ca/cic/datasets/ddos-2019.html

[59] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "Towards a reliable intrusion detection benchmark dataset," *Software Networking*, vol. 2018, no. 1, pp. 177–200, 2018.

[60] R. Doriguzzi-Corin, "Lucid dataset parser," https://github.com/doriguzzi/lucid-ddos/blob/master/lucid_dataset_parser.py.

[61] Open Networking Foundation, "Performance of bmv2," 2019, [Accessed: 30-Jun-2023]. [Online]. Available: https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md

[62] P. Jurkiewicz, G. Rzym, and P. Boryło, "Flow length and size distributions in campus internet traffic," *Computer Communications*, vol. 167, pp. 15–30, 2021.

[63] Imperva, "LOIC," [Accessed: 31-Oct-2023]. [Online]. Available: https://www.imperva.com/learn/ddos/low-orbit-ion-cannon/

[64] Imperva, "HOIC," [Accessed: 31-Oct-2023]. [Online]. Available: https://www.imperva.com/learn/ddos/high-orbit-ion-cannon/

[65] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers & Security*, vol. 31, 2012.

[66] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," in *Proc. of ICISSP*, 2018.

[67] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization." *ICISSp*, vol. 1, pp. 108–116, 2018.