POLITECNICO
MILANO 1863

School of Industrial and Information Engineering

Master of Science in Telecommunication Engineering

Electronics, Information and Bioengineering Department

Traffic classification offloading to stateful data plane in Software-Defined Networking

Supervisor:

Prof. Antonio CAPONE

Advisor:

Eng. Davide SANVITO

Graduation Thesis of:

Daniele MORO

ID 837957

Academic Year 2015 - 2016

Contents

List of Figures

List of Tables

Abstract

Traffic analysis is currently used by Internet Service Providers (ISP) to gain important insights on users' behavior, and to develop from them new applications that can best exploit their network. The volume of encrypted traffic is increasing and this poses new limits on the potentiality of Deep Packet Inspection (DPI) techniques, normally used to analyze traffic flows. However, an important amount of information can still be extracted from the first packets belonging to a connection which usually are transmitted in clear. Recent research works have shown that traffic inspected by the DPI can be reduced without losing classification accuracy. In this thesis we propose to exploit stateful SDN data plane to offload, down to network elements, the process of filtering. We show that it is possible to dramatically decrease the amount of traffic analyzed by the DPI with zero-classification accuracy loss. We also show that we can reduce the computational requirements of the DPI and that the impact of the functions offloaded to network switches is negligible in terms of their performance. By taking advantage of the programmability of the data plane we also managed to delegate to the switches the process of statistics collection (such as per-flow number of packets, number of bytes, and duration), that otherwise would be lost by applying our filtering scheme. We gave evidences that this solution can be implemented in hardware, and also discuss an alternative implementation, based exclusively on a stateless data plane. Finally, we identify additional extensions to further optimize the solution.

Sommario

L'analisi del traffico viene utilizzata dagli Internet Service Providers (ISP) per acquisire importanti informazioni riguardo il comportamento degli utenti. Queste informazioni vengono sfruttate dagli ISP per sviluppare nuove applicazioni e per sfruttare al meglio la rete in loro possesso. Il continuo aumento del traffico criptato crea sempre nuovi limiti alle tecniche di Deep Packet Inspection (DPI) usate per analizzare i flussi di dati. Una quantità importate di informazioni può comunque essere estratta dai primi pacchetti della connessione, i quali, di solito, vengono trasmessi in chiaro. Recenti lavori di ricerca hanno mostrato che è possibile ridurre il traffico ispezionato dalla DPI senza perdere precisione nella classificazione. In questo lavoro di tesi proponiamo di sfruttare il data plane SDN stateful per delegare il processo di filtraggio agli elementi di rete. Abbiamo mostrato che è possibile ridurre drasticamente le necessita computazionali della DPI e che l'impatto delle azioni delegate al data plane è trascurabile. Sfruttando le potenzialità del data plane programmabile siamo anche riusciti a demandare agli switch il processo di collezione delle statistiche dei flussi (es. numero di pacchetti, numero di byte, durata) che altrimenti sarebbero state perse con il nostro sistema di filtraggio. Abbiamo anche dato prova che la nostra soluzione può essere implementata in hardware. Inoltre, abbiamo proposto un'implementazione alternativa basata esclusivamente su data plane stateless. Infine, abbiamo identificato ulteriori estensioni che potrebbero ottimizzare ulteriormente la soluzione proposta.

Chapter 1

Introduction

The adoption of encrypted connections over HTTPS has undergone a remarkable boost in the last few years: in 2016 more than 60% of Internet traffic was encrypted [1], and this trend is still increasing [2]. While encryption used to be confined to handling sensitive transactions (involving the exchange of a small volume of traffic), nowadays HTTPS is instead normally used also for carrying multimedia content, including videos (for instance Facebook and YouTube have enabled HTTPS by default in 2013 and 2014, respectively). To further push this trend, in 2015 Google announced that sites served through HTTPS would be ranked better [3]. At the same time, it is known [3] that Google crawlers still attempt to parse HTTPS equivalent of HTTP pages. Lastly, not only there is an ongoing increasing trend of privacy concerns among Internet users [4], but also HTTPS decreases risks for users browsing on websites vulnerable to content injection attacks. All these reasons together have made the spreading of secured, encrypted protocols be even more relevant these days.

However, the increased volume of encrypted traffic makes in-network traffic analysis very challenging. As a matter of fact, the tools normally used to perform Deep Packet Inspection (DPIs) have strongly relied on the possibility to parse packet payloads transferred in clear text. After all, this does not

mean that the traffic is undetectable or unidentifiable, it just means that the transferred content is private.

At the same time, in the preamble of many encrypted protocols, such as HTTPS, the two endpoints need to mutually exchange keying materials (i.e. the certificates), as well as choose the parameters of the encryption algorithm to use in the secured part of the connection. However, normally, the dialogue in this initial phase is performed in clear text, making it possible, to packet inspection tools, to extract important data to characterize the forthcoming encrypted flow. Furthermore, when resolving Domain Names, each device sends DNS requests and receives DNS replies, and also these packets are sent in clear. With DNS and TLS certificate data, most of the DPIs can perform flow classification with encrypted traffic as well.

Usually DPIs (and in particular software-based DPIs) have very high computational requirements in terms of CPU and memory consumption since they need to analyze traffic at wire-speed. On top of that, very often DPIs also create new issues at the network level, since the traffic needs to be mirrored and forwarded to the DPI node for inspection. For both these reasons, to keep the running costs limited, the state-of-the-art deployment of the deep packet inspection functionality in a telco infrastructure is such that *potentially* any subscriber can be subject to inspection, but in practice, the allocated peak-capacity can only handle a portion of the overall population. Therefore, in most of the cases, the DPI is activated only on a sampled set of subscribers.

The objective of this work is to go one step beyond these drawbacks, making it possible to perform the traffic inspection functionality, at a fraction of the overall costs. The rationale behind this work is to send to the DPI node only the portion of the traffic that the software needs to analyze to make the correct classification, by offloading precious computational as well as network resources. The solution proposed in this thesis can be adopted as a switch-based stateful SDN technique to potentially scale the DPI functionality to the

entire population of a telco, in a cost-effective manner. As a matter of fact, extensive numerical analysis of our proposal have shown that it is possible to drastically reduce the quantity of traffic directed to the DPI (up to 99% as measured in our experimental results), without loosing classification accuracy and traffic flow characteristics, by offloading most of the job to a stateful switching data plane.

Other attempts to solve this problem have been proposed in the literature. In [5], the authors have studied the impact of traffic filtering, showing that the DPI can still classify the traffic with negligible accuracy loss. However, the authors did not propose techniques to make filtering possible without losing fundamental information like flow statistics (duration, exchanged volume, round-trip time and others), that nowadays are calculated directly on the DPI node itself. With SDN, it would be possible to implement the filtering and statistics collection functions, making the DPI and the SDN controller cooperate together. In one such solution, the DPI would contact the controller when the classification of a flow has been completed, thus a new rule on the switch could be installed to stop forwarding to the DPI. However, with traditional SDN the switch cannot take any decision without being triggered by the controller. This would not only introduce latency between the completion of the flow classification (done on the DPI), and the installation of the rule in the switches, but it would also significantly increase the computational complexity on the controller.

Our approach is different and goes beyond these limitations by taking advantage of stateful data plane to delegate filtering and statistics collection to the fast path without the need of the controller to install the filtering rules. The controller needs only to collect the information about flow characteristics provided by the switches.

The thesis is organized as follows:

- Chapter 2 provides an overview on SDN and current state of the art DPIs, discussing classification methods used in DPI, stateful data planes and application aware SDN;

- Chapter 3 describes the adopted solution, the implementation, possible extensions and some particular network use cases where this solution can fit best;

- Chapter 4 is about the performance evaluation of the solution described in the previous chapter, it discusses advantages and disadvantages of the solution and it sketches an alternative implementation that does not require a stateful data plane;

- Chapter 5 concludes the manuscript and gives an outlook on future works and improvements of this solution.

Chapter 2

State of the Art

2.1 Deep Packet Inspection

2.1.1 What is DPI

Deep Packet Inspection (DPI) is a network functionality that involves analyzing packet payloads and headers, extracting additional parameters to characterize the ongoing flow, with the aim of making some decisions (e.g. dropping a packet, rate limit a flow, generate alerts etc...) based on the inspection result, or simply for collecting statistical data.
DPI entails many Internet technologies such as firewalls, packet capturing or sniffing techniques.
DPI is an *enabling technology* [6]: it is a generic capability that supports many different applications or use cases. As an evidence of this fact, usually commercial and open source DPIs provide a SDK, to develop your application exploiting the SDK capability of traffic recognition and classification.
The main function of DPI is *recognition* of characteristics hidden in the traffic, but it can eventually do also *manipulation* and/or *notification*.
Recognition involves the detection and identification of traffic, analyzing and

inspecting flows in real time. It can be used to detect protocols, applications, viruses, malware or other special format of data.
Manipulation is the active intervention in a live traffic stream, to optimize, shape or control it, eventually triggered by the recognition results. Usually DPI vendors will supply and maintain the signatures that enable recognition, while their customers define the rules for manipulating the recognized applications.
Notification is an indirect form of intervention: it regards the generation of statistical reports, the issue of alarms or the generation of bills with respect to actions triggered for the manipulation. These actions are not executed in real time, they can be done later on and thus they do not need to directly act on the network.

An important clarification to make, regards the "deep" part of the term DPI: analyzing and decrypting the payload of packets does not mean that DPI breaks the confidentiality of communications. DPI can work well also with encrypted traffic, because it is able to exploit essential information also from headers, TLS handshake and DNS. Clearly, in this case the DPI can not extract valuable information such as the precise page that the user is viewing or the specific video is watching, but the authentication, the privacy and the integrity of the encrypted connection is guaranteed.

2.1.2 DPI Application Scenarios

Providing a classification of DPI application is a difficult task as there are multiple use cases that can be mixed up. According to [6], [7], [8] we can identify 5 groups of use cases, namely network visibility and bandwidth management, user profiling, network security, copyright policing, government surveillance and censorship.

1. *Network Security*: this was the original use case for DPI as it was initially

developed for Intrusion Detection Systems (IDS). DPIs are able to detect known malware or known pattern of attack and, when used for network security, combine IDS and IPS (Intrusion Prevention System), to take measures to prevent possible threats.

Recently preventing data from leaving the intranet has gained attention. This specific practice is called Data Loss Prevention (DPL) and also in this case DPIs are fundamental tool to identify sensitive information hidden in normal streams and block them from leaving the local network. In this case the confidentiality has to be broken: an enterprise willing to implement a DLP system has to integrate in its network a SSL proxy server, that act as a "man in the middle" in the HTTPS connection allowing DPIs to inspect the entire content of the communication

2. *Bandwidth Management*: ISPs have a strong motivation to deploy DPI for bandwidth management. Bandwidth management is the process of controlling the traffic on a network link to avoid poor network performance or network congestion. This technique relies on traffic classification in order to classify traffic into different categories and then apply mechanism such as traffic shaping, scheduling algorithms to each class of traffic differently.

   Typical use cases include prioritizing real-time interactive applications (e.g. VoIP calls, online gaming), rate limiting bandwidth-intensive applications (e.g. P2P, direct download) and blocking undesired applications (e.g. P2P in enterprise environment). In conclusion DPIs in this field are used to ensure best Quality of Experience for all users.

3. *User Profiling*: Subscriber awareness is also an important field of application of DPIs. Profiling can allow network operators to tailor their service offers to the user's specific behavioral patterns, providing, for example, zero-rated applications or services.

   "Ad Injection" can be another use case of this category, ISPs can lever-

age their direct knowledge of websites visited by their customer to make targeted advertisement by injecting them into web traffic. However, this technique is controversial because it may affect the customers' browsing experience, privacy and security.

4. *Copyright Enforcement*: DPI for copyright inspection can not employ traditional recognition techniques because it should be able to recognize the fragments of copyright contents. For example Audible Magic [9] provides fingerprinting technology to identify protected material: a DPI application can calculate the fingerprint and match this one with the database of registered materials.

5. *Government Surveillance and Censorship*: DPI can make anything that happens on a network visible and recordable to governments. Government content censorship is another DPI-based application that blocks illegal content flowing in the Internet; due to difficulty and high cost of real content-based censorship, URL blocking has been the most common form of censorship so far.

Another responsibility of a DPI is flow statistics collection. DPIs typically process the whole traffic so it is simple for them to calculate flow-based statistics such as time duration, transferred bytes, packet interarrival time and others. Not all the use cases listed above need these type of statistics, but surely those info can be used by ISPs to increase the knowledge about the services used by their users, to implement zero-rating or other kinds of smart billing or to use this information to make fraud detection [10].

A final classification that can be made about DPI application is related to the timeline of the classification. In case of *Bandwidth Management* or *Network Security* it is needed to have real time classification but, dealing with *User Profiling* it is not, as it is only necessary to collect data and analyze it later on. The same applies to flow statistics: some apps need these statistics in real

time, while for some others offline statistics are enough.

2.1.3 DPI classification method

This section presents different methods to perform Deep Packet Inspection.
The general concept of DPI covers the inspection of both the packet header and payload content. The headers can be used for protocol and application classification while the payload can be used for content-based recognition (for example you can define some attribute that can be pulled out from the payload such as YouTube video ID, Facebook Profile or the web-search you are doing in Google Search).
As mentioned in [7] and [11], 4 main DPI techniques can be identified:

- *Port-based approach*: it is the most common and traditional method for traffic classification. It uses the well-known TCP/UDP port numbers assigned by the IANA to popular protocols.
  This classification method is considered inaccurate as application like P2P or passive FTP use random ports or ports assigned by other protocols. Currently it is still employed for identifying applications which use preassigned ports.

- *Statistical approach*: this approach uses information such as ports number, packet length, transport layer protocol, inter-arrival time of packets, flow start-stop time and others to characterize the traffic and estimate which application or protocol the traffic may belong to. Machine learning techniques combine statistical methods and heuristics-based algorithms for modelling and analysis. Statistical analysis has an higher classification accuracy than port-based method and is insensitive to packet payload encryption.

- *Pattern matching*: through DPI it is possible to compare the contents

of captured packets against a set of rules, represented by regular expressions.

This category of techniques need fast algorithms for pattern matching

- *Protocol decoding*: it is a lightweight version of the pattern matching techniques. It recognizes protocols by looking at the characteristics of the protocol headers (magic numbers, incrementing counter, session identifiers, etc.) and packet sequences. Protocol decoding can achieve high accuracy with low false negative rate but it requires a deep understanding of the protocols.

When the traffic is not encrypted, all the previous methods can equally be utilized for the inspection. On the other hand, when the traffic is encrypted, only data harvested from DNS requests/responses and TLS certificates exchanged in the preamble of the connection can be used as input data to do the classification.

2.1.4 DPI optimization techniques

To optimize DPIs we can choose among two main options: (1) by optimizing the classification algorithm itself; or (2) by filtering the amount of traffic that the DPI node should inspect. The first type of optimization is based mainly on Regular Expression and State Machines optimization [7]. The scope of this work is instead the second class of optimizations, and therefore the approach will be filtering the amount of traffic sent to the DPI node.

As we can see in [5] the authors have analyzed the impact of reducing the number of per flow packets and the quantity of per packets bytes of the traffic analyzed by the DPI. They first analyzed the so called *Snapshot-based classification* where they variate the quantity of bytes analyzed for each packets and they observed the variation in accuracy and processing cost. According to

their results an hard limit of 256 bytes per packet is a good trade off between the improvements in processing costs and the impact on accuracy. Finally they analyzed the approach of limiting the number of packets per flow. They concluded that a limit of 2 analyzed packets is sufficient to have good accuracy. While the values obtained in [5] are very small, those depend on different factors, including, but not limited to, the traffic trace itself, the DPI used, the desired flow classification accuracy, the type of online services used.

A similar approach was also adopted in [12], where the authors reduced the number of collected packets and they truncated also the packets' payload collecting only the first bytes. Interestingly by limiting the number of analyzed packets per flow they reached 100% accuracy with only 7 packets per flow. The result obtained are justified by the fact that most applications exchange control messages are at the beginning of the communication session, so the most important part of the flow, from the DPI stand point, is the first part of the session. Regarding the truncation of packet payload, according to the results in [12] a value between 700 and 800 bytes will provide a good balance between accuracy and performance. Another interesting analysis done in this work, regards the application classes classification changes: they evaluated the quantity of traffic that changed classification after the optimization cited before. They obtained that the streaming class was the one that suffers more from classification changes when the packet payload is truncated to 200 bytes. In this work they extensively evaluated also the impact of filtering and truncating on the processing time. They obtained that by analyzing only the first 7 packets of a flow, processing time can be reduced around 70%.

In [13] the authors investigated the sampling of traffic being monitored, by analyzing (as the papers before) two sampling techniques: per-packet payload sampling and per-flow packet sampling. In particular they customized OpenDPI (an open source DPI, no more supported but substituted by nDPI [14]) to be able to parse only a specified length of bytes within each packet's

payload. They used as ground truth the classification given by OpenDPI with the whole packets database, removing the unclassified flows, then they made the classification with the modified version of OpenDPI and they compared the number of correctly classified flows with respect to their ground truth. The most interesting result regards the per-flow packet sampling: also in this case they used a customized version of OpenDPI, where the classification part is skipped for the packets whose ordinal number is larger than a fixed threshold. In this case they reached 99% classification accuracy with only 10 packets per flow. In this last evaluation, they also clarified that the sampling speed-up comes only from speeding-up the flow classification itself, but there is no gain in the operation of mapping packets to flows, as this operation is independent. Based on this observation, with the solution proposed in this thesis we can obtain a further gain because we are able to skip also the mapping of packets to flow, since this phase can be performed inside the switch itself.

All the works just mentioned proposed interesting packets filtering method but did not find the most efficient way to implement it. In [12] it is stated that the filtering can be done inside the same machine running the DPI, deploying a mechanism that avoids the forwarding of packets from kernel to user space, this surely increases the performance but it is not the best way to do such a filtering. In the third work they modified OpenDPI for the filtering task, also in this case, the filtering part is not optimized. According to our findings a filtering technique applied directly on networks element will be definitely more optimized, saving both computational power and valuable network resources In this case the drawback is that the DPI can not collect directly the statistics of flows, but in the following chapters we will present an approach able to overcome also this issue.

Another approach is described in [15] where the authors attempted to exploit stateful data plane, in particular *OpenState* [16], to make the filtering operation directly inside a network element. In Section 2.2 of this thesis we provide

an in-depth background on SDN and stateful data plane. In this work the authors implemented a packet counter directly inside the switch. By counting the packets, they can send to the so called Traffic Classifier (TC) only the first valuable packets of each flow. They also implemented a countdown interruption technique used by the Traffic Classifier to interrupt the mirroring of packets when the classification is completed. They did not evaluate the classification accuracy in an extensive way nor the impact on the TC of the filtering technique implemented, but they made some evaluation on the memory occupancy on the switch of their implementation. They also proposed a comparative implementation on classic OpenFlow, where all the traffic is sent to the controller which is in charge of the counting process itself; the controller will then install a rule on the switch as soon as the countdown reaches 0. They made a comparison between this implementation and the one with stateful data plane and it is clear that the stateful data plane helps to reduce the traffic sent to the Traffic Classifier. However, the approach described in [15] could eventually be further improved by removing the forwarding of the first packet of each flow to the controller, a feature natively supported by OpenState, and make the switch completely autonomous for the task of filtering, but it was not done.

As already mentioned all the works analyzed only focused on the filtering part without considering the drawbacks of this task in the flow statistics collection. Talking about the software DPI engine itself, some of them implements techniques to do a sort of filtering, offloading the traffic from the main classification engine when the classification is completed. Doing this offload inside the DPI machine allows the DPI itself to continue collecting statistics of the already classified flows with a performance gain but without any network resource savings.

With this work we are going to present a solution to solve the problems found making both filtering and statistics collection directly inside the switch having

computation gain in the DPI and network resource savings.

Until now we presented works that used only payload inspection based method, there are, however, other works that focused on early traffic classification using Machine Learning techniques. For instance in [17] the authors proposed a method that needs only the size of the first packets of a flow and their direction to classify the application. It is based on *Unsupervised Learning* (clustering). It identifies the clusters based on the packet size and then it assign a label to each clusters using a DPI. These trained model is then used to classify the traffic. They reached high classification accuracy (on their test set they reached more than 90% accuracy) using only the size and direction of the first 5 packets.
In [18], instead, the authors used *Supervised Learning*, in particular Support Vector Machine, to make flow classification. Also in this work they reached high classification accuracy with only 5-6 packets per flow using only the dimension of that packets. Both works have the objective of making early traffic classification, ideally choosing the flow class directly in the first 5-6 packets of each flows. Our method focused on traffic filtering and statistics collection, but it can be exploited also for these type of statistical classification.

2.2 Software Defined Networking

2.2.1 What is SDN

Software Defined Networking (SDN) is an innovative approach to programmable network. SDN consists of decoupling the control and data plane of a network [19]. The main advantage of SDN is that it enables innovation and flexibility: by having access to a software component to manage the control plane, new use cases becomes possible. The main functionality of a switch is forwarding packets according to a set of rules. In SDN these rules are managed by a soft-

ware running in a logically centralized node: the controller. Switches expose a programmable interface which allows a software to configure the forwarding behavior. Developed applications can control the switches by running on top of a Network Operating System, which works as an intermediate layer between the switch and the application. Network Operating System is a key concept in SDN. It comes from the idea of abstracting the complexity of the underlying network: as an operating system abstracts the underlying hardware components, the NOS hides the complexity of the topology and the network devices. The Network Operating System is responsible for the abstraction provided by SDN to its users. With SDN, part of the complexity of the network is moved to the software-based controller which has a global view of the whole network.

One important feature of SDN is its ability to provide a network wide abstraction as a Platform-as-a-Service (PaaS) model for networking. The underlying physical network and the topology are hidden to the user; the network is abstracted and presented as a single switch and users can define forwarding policies reasoning on top of the simplified network view.

One of the proposal to standardize the communication between the switches and the controller is OpenFlow [20]. The authors identify that it is difficult for the networking research community to test new ideas in current hardware. The source code of the software running on the switches cannot be modified and new network ideas cannot be tested in realistic traffic settings. OpenFlow provides a means to control the switch without requiring the vendors to expose the code of their devices. OpenFlow networks have specific capabilities; it is possible to control multiple switches from a single controller; it is also feasible to analyze traffic statistics in software. Forwarding information can be updated dynamically as well, and different types of traffic can be abstracted and managed as flows. Thanks to its pragmatic balance today OpenFlow has become the de-facto standard in SDN: on one hand, it allows researchers to run experiments on heterogeneous switches in a uniform way at line-rate; on the

other, vendors do not need to expose the internal workings of their switches.

2.2.2 OpenFlow

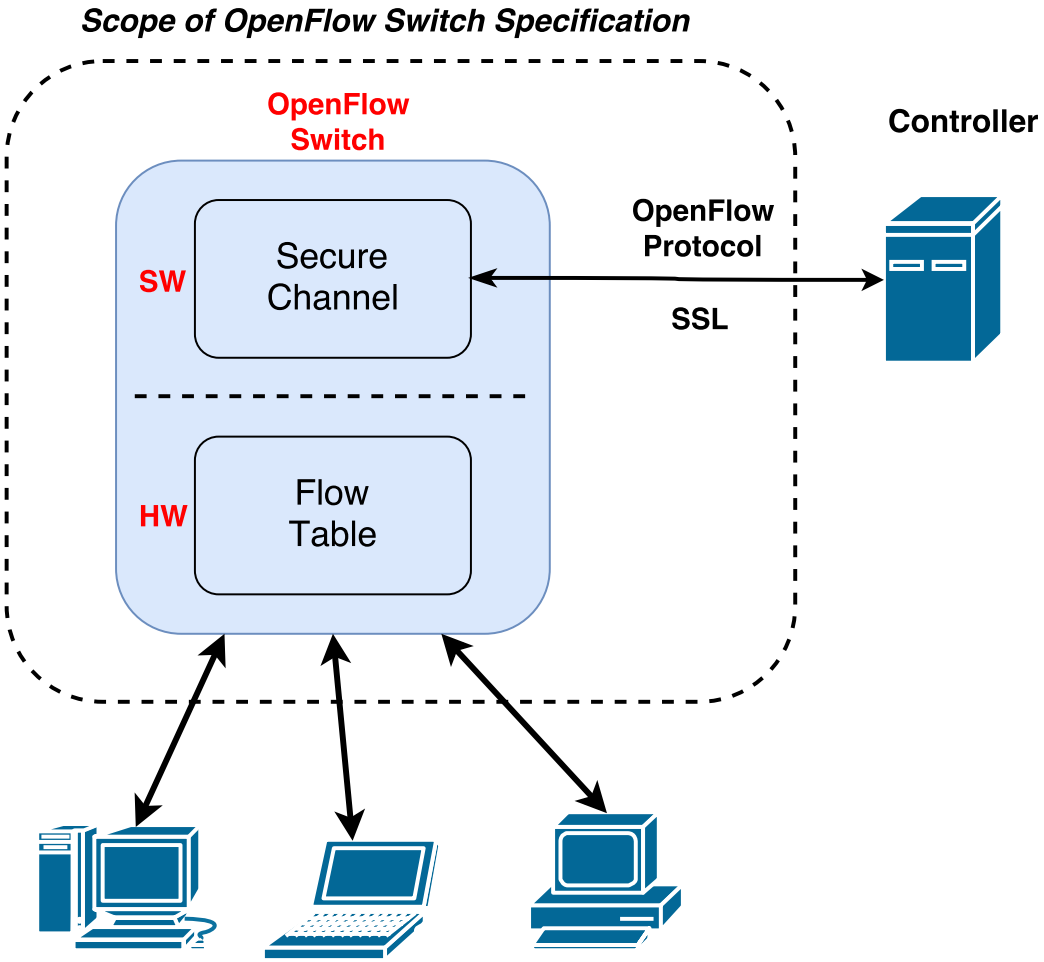

Figure 2.1: OpenFlow Switch, the flow table is controlled by a remote controller

In this section we are going to make a brief description of OpenFlow, primarily focusing on the OpenFlow architecture and on the processing of packets. OpenFlow creators exploited the common set of functions of the routing table contained in most of the modern Ethernet switches and routers where routing table are built from TCAMs: these tables process packets at line-rate and are used to implement some specific forwarding inside the networking element. OpenFlow provides an open protocol to program these tables.

An OpenFlow architecture consists of a Flow Table, a controller and a secure channel as show in Figure 2.1. Switches use the flow table to forward packets and expose it to the controller as a match-action table. Each entry specifies a treatment (expressed with a list of actions) for a certain type of traffic (given by a set of match fields). For example a rule can be configured to drop all the packets coming from a certain TCP port blocking all the traffic of this specific port. Incoming packets are compared with the match fields of each entry and if there is a match, the packet is processed according to the action contained in that entry. In addition, flow entries contain counters used to keep statistics about packets. The packets can be also encapsulated and sent to the controller.

The controller runs application programs responsible for manipulating the switch's flow table using the OpenFlow protocol.

The secure channel is the interface between controller and switch: through this channel the controller manages the switches, receives packets from the switches and sends back packets to them.

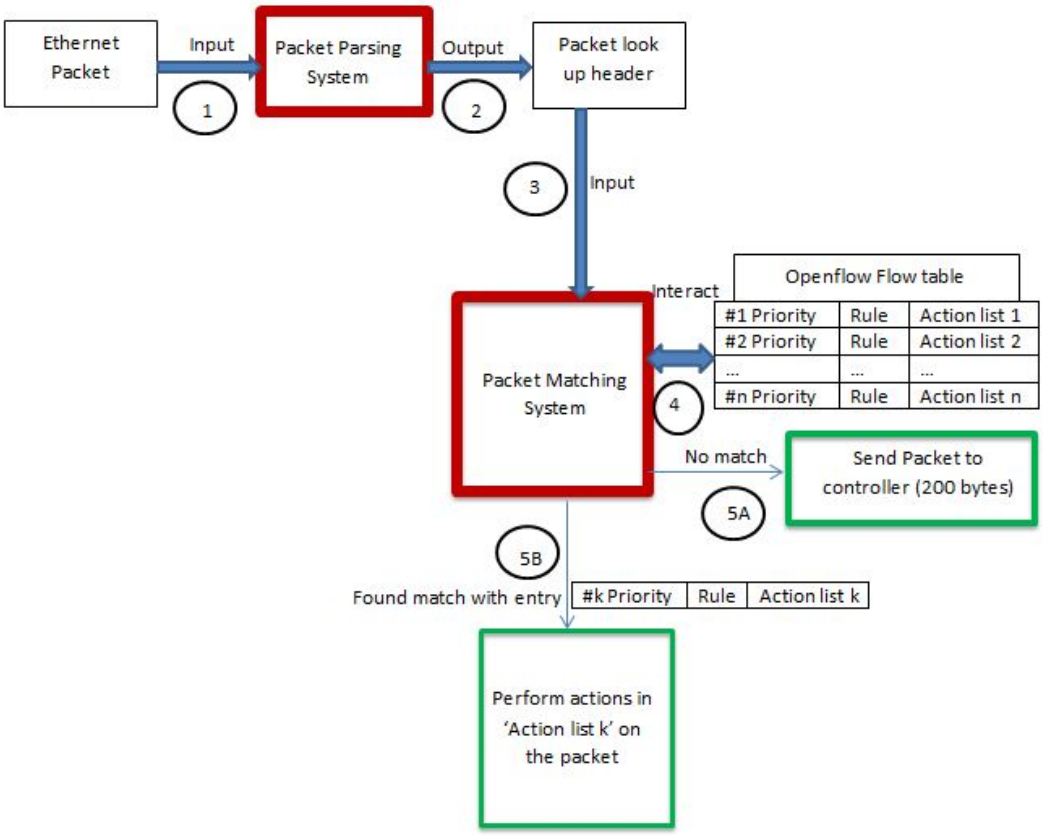

Figure 2.2: Packet processing and forwarding in an OpenFlow 1.0.0 switch

Figure 2.2 show the details of the data plane processing an Ethernet packet in an OpenFlow 1.0.0 switch. In step 1 the Ethernet packet entering the switch goes to a packet parsing system. In step 2, the header fields are extracted and placed in a packet lookup header. In step 3, the packet lookup header generated is sent to the packet matching system. In step 4, the packet lookup header is compared to the rules defined for each flow entry in the OpenFlow flow table. Each flow entry has a priority that defines the order in which the matching is performed. If a match is found, the actions specified in the matched flow entry are performed on the packet (step 5B). Otherwise the packet is sent to the controller (step 5A) which can forward it to the destination and/or install a new flow table entry.

Newer version of OpenFlow introduced the concept of group table (common table to which a flow entry can redirect packets), added multiple flow tables,

increased the set of matching fields and introduced instructions instead of actions (instructions are more complex and they include modifying a packet, updating an action set or updating metadata). In the latest versions of OpenFlow, the packet processing procedure has slightly changed. When the packet enters the switch, it is sent to the first table to look for the flow entry to be matched. If there is a match, the packet gets processed there and, eventually, if there is another table that the particular flow entry points to, is then sent to other flow tables. This happens until a particular flow entry does not have reference to any other flow table.

2.2.3 Stateful data plane

A relatively recent trend in SDN regards stateful data plane. This term means that we are able to express forwarding policies which depend on the previous history of a flow. With OpenFlow the switch becomes "dumb" with all the smartness in the controller, while now we are going to transfer some of this "smartness" inside the switch itself. It may be objected that the dumbness of the switch is in line with the vision of SDN's control and data plane separation. This is true, however several stateful tasks, just involving local flow states, are unnecessarily centralized in the controller only because OpenFlow does not allow to deploy them directly into the switch.
Another drawback of classic OpenFlow regards the delay from the contact of the controller and the deployment of a rule in the switch due to processing at the controller and network transfer of control messages. Centralization of the network applications' intelligence may not be an issue whenever changes in the forwarding states do not have strict real time requirements, but for applications which rely only on local flow/port states, the latency imposed by the reliance on an external controller rules away the possibility to enforce software-implemented control plane task at wire speed.

In the next section we are going to talk about Open Packet Processor (OPP) [21], that is one of the example of architecture of a stateful data plane implementing the Extended Finite State Machine (EFSM) abstraction.

2.2.3.1 Open Packet Processor - OPP

The Open Packet Processor (OPP) [21] is a programmable data plane architecture, implementing an Extended Finite State Machine (EFSM) [22].
It allows to process packets in a stateful fashion directly on the fast path while the packet is traveling in the switch pipeline; efficient storage and management of per-flow stateful information allows to specify and compute a wide class of stateful information.

EFSM formal notation		***Meaning***
I	input symbols	all possible matches on packet header fields
O	output symbols	OpenFlow-type actions
S	custom states	application specific states, defined by programmer
D	n-dimensional linear space $D_1 \times \cdots \times D_n$	all possible settings of n memory registers; include both custom per-flow and global registers
F	set of enabling functions $f_i : D \to \{0, 1\}$	Conditions (Boolean predicates) on registers
U	set of update function $u_i : D \to D$	Applicable operations for updating registers' content
T	transition relation $T : S \times F \times I \to S \times U \times O$	Target state, actions and register update commands associated to each transition

Table 2.1: Extended Finite State Machine model

The abstraction implemented by this architecture is the Extended Finite State Machine model: the evolution of the forwarding is described by a state machine where each state defines a forwarding policy and state transitions are triggered by packet-level events, time-based events or evaluations of conditions (computed on flow state and/or packet header fields).

As summarized in Table 2.1 this model is specified by means of a 7-tuple $M = (I, O, S, D, F, U, T)$. Input symbols I (OpenFlow type matches) and Output Symbols O (actions/instructions) are the same as OpenFlow. S is a finite set of states that let the programmer freely specify the possible states in which a flow can be in relation to the desired application. With this model the programmer can define custom (per-flow) registers and global (switch-level) parameters. All these registers are summarized in the EFSM model via the array D.

The *enabling functions* $f_i : D \to \{0, 1\}$ serve the purpose of making comparison where the input can be selected by the programmer. The output of this functions can trigger a programmed state transition.

Moreover, this model allows the programmer to update the content of the deployed registers using the *update functions* $u_i : D \to D$ that can be arithmetic or logic primitives.

The actual computation step in an EFSM is the transition relation $T : S \times F \times I \to S \times U \times O$, which is a map between some type of input symbols, conditions on register and application specific state to a new state, an update of register and an output action (OpenFlow standard action).

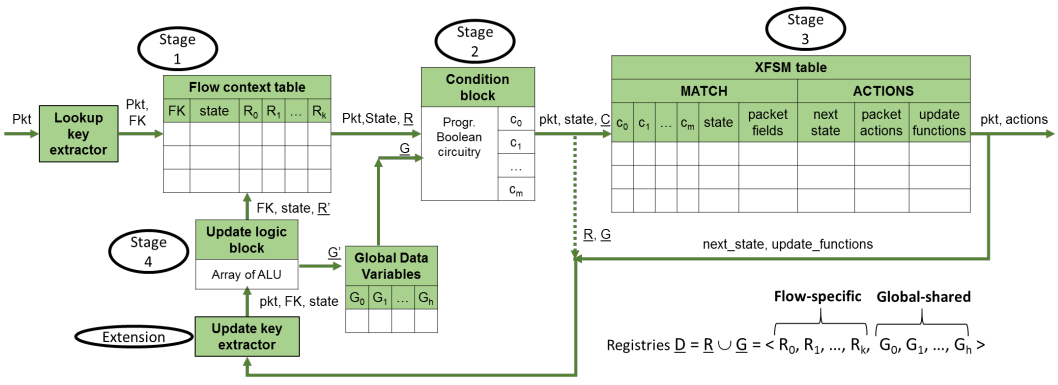

Figure 2.3: OPP architecture

Regarding the OPP architecture, it can be summarized in 4 stages, as illustrated in Figure 2.3

- *Stage 1: flow context lookup.* In this stage a *Flow Identification Key*

(FK) is extracted from the packet. FK identifies the entity to which a state may be assigned. The desired FK is configured by the programmer and depends on the specific application. The FK is used to extract the *flow context* from the *Flow Context Table*, which is expressed in terms of the state label s_i currently associated to the flow and an array $\vec{R} = \{R_0, R_1, \ldots, R_k\}$ of *k+1* registers. The extracted flow context is appended as metadata to the packet and it is forwarded to the next stage.

- *Stage 2: conditions' evaluation.* In this stage the programmer-specific conditions are computed. They can take as input either the per-flow register values, as well as global register delivered to this block as an array $\vec{G} = \{G_0, G_1, \ldots, G_h\}$ of *h+1* global variable. This stage represents the computation of the *enabling functions* specified by the EFSM abstraction. The conditions can be extended also to packet header fields. The output of this block is a Boolean vector $\vec{C} = \{C_0, C_1, \ldots, C_m\}$ which summarizes if the i-th condition is true of false.

- *Stage 3: EFSM execution step.* The vector obtained as output of the previous step, along with the state table and the necessary packet header fields are used as input of the EFSM table which is implemented as a TCAM. This table performs wildcard matching (so can be defined a "don't care" for some specific inputs). Each TCAM row models one transition in the EFSM and returns a 3-tuple including (1) the next state in which the flow shall be set, (2) the actions associated to the packet, (3) the information needed to update the registers.

- *Stage 4: register updates.* The role of this stage is to perform arithmetic processing on the registers using as input the information available at this stage (previous values of the register, information extracted form the packets, etc.).

- *Extension: cross-flow context handling.* Some states for a given flow can be updated by events occurring in different flows. This functionality is useful to generalize the EFSM abstraction by permitting the programmer to use a Flow Key during lookup and use a possible different Flow Key for updating a state of a register. The simplest example is MAC learning.

In [21] authors also demonstrated the hardware feasibility of the OPP architecture with a netFPGA.
The authors also developed a complete OPP virtual software prototype [23] for both the switch and the controller. The software implements the OPP architecture described above, as an extension of the OpenFlow protocol: the controller application can configure a stateful forwarding by means of the OPP protocol to populate the EFSM table entries and to configure conditions, functions, key extractors and initial global register values.

2.2.4 SDN application awareness

Application-Aware Networking is a promising approach to provide good application quality to users in scenarios with limited network resources. Merging SDN with application awareness can provide better QoE (Quality of Experience) and enhance performance of specific applications. Behind this idea there is the need to classify applications so that the forwarding can be aware of the type of traffic flowing and take actions consequently.

As can be seen in [8], where DPIs are considered as the mean to make classification, knowing the application of network flows will create a virtuous circle in which real-time information is fed back to the SDN controller to allow continuous adjustment to circumstance, optimizing both the efficiency with which resources are consumed and the quality of the end-user experience. In a dynamic service environment, where cloud and virtualization technologies are mainly used, real-time analytics will help providers to allocate resources dy-

namically and in an optimized way. Moreover, the knowledge of applications used can activate service differentiation to shape traffic according to network capacity.

There are also other works in this direction. In [24] authors explored the possibility of optimizing YouTube streaming to improve the QoE (that is measured as function of the number of stalls in the video stream). They tested a solution in which they identified the YouTube traffic using DPI analysis and then prioritized it exploiting OpenFlow. They concluded that the users can benefit profoundly from the application awareness approach.

In [25] the authors presented a framework which incorporates application awareness into SDN. They used Machine Learning classification techniques and they developed a technique to make data collection to feed their classifier. Using these techniques they incorporated L7-awareness into SDN, making available this classification to any SDN application that need L7 knowledge.

Also in [26] they worked in the direction of application-layer classification in SDN, they combined ML and DPI techniques in a *MultiClassifier* to achieve high classification accuracy and fast computation. They also thought of making classification available as an API for some application that want to exploit them. In the work they mainly analyzed the MultiClassifier structure defining threshold to switch between ML and DPI classification to reach better performance. At the end they also analyzed the fact that their architecture, that involves OpenFlow to steer traffic to the MultiClassifier passing through the Controller, brings too much traffic to the controller creating scalability issue.

Following this vision, DPI and SDN are really coupled fields; finding a way to optimize the DPI computation using SDN can represent a good direction to explore.
SDN can optimize the DPI computation requirements and the output from DPI can, in turn, help optimizing the network usage.

Chapter 3

Problem Description and Proposed Solution

3.1 Problem Description

The objective of this work is to reduce the DPI impact on computational and network resources, while ensuring industry-standard classification accuracy, solving the shortcomings extensively discussed in Section 2.1.4. In order to do that, the rationale of our proposal is reducing the quantity of traffic delivered to a DPI while at the same time moving to the network switches all the functionalities that the DPI cannot perform when it does not receive all the traffic.

First we need to choose our definition of flow. We can adopt the same definition of flow used by DPI engines: the 5-tuple IP source address, TCP/UDP source port, IP destination address, TCP/UDP destination port, IP proto.

To offload the DPI we will only send it the part of the traffic strictly necessary to handle the classification: we will therefore implement a per-flow sampling, setting a threshold on the number of packets per flow forwarded to the DPI

itself. For classification purposes, and given the increase in encrypted traffic, only the first initial packets at the beginning of each flow will be used by the DPI. For HTTPS in particular, the first few packet contains TLS handshake with Certificate Exchange from which DPI can extract, for example, the *Common Name* that identifies the host name associated to that particular certificate.

We can think of a sampling technique that filters out all traffic except the first packets of a connection (where most of the information is placed).
This type of filtering, however, creates an undesired effect in the DPI engine: since the DPI does not receive all the traffic, it cannot calculate flow related statistics such as bytes transferred or duration metrics that are of vital importance for a lot of applications especially for network analytics.
These problems create a need for an efficient solution to make both filtering and collection of traffic statistics inside network elements.

3.2 Proposed Solution

This section illustrates our proposed solution to efficiently offload filtering and statistics collections down to the network, exploiting stateful data plane. We choose OPP as stateful data plane model (described in Section 2.2.3.1) because it provides an easy and flexible abstraction based on Extended Finite State Machine, suitable to describe a flow session. Moreover, with OPP we can have registers associated to flows. These are fundamental features to collect flow statistics.

Before diving into the implementation in SDN switches, we want to describe our stateful processing. As the first implementation of such a filtering scheme, we focused mainly on TCP traffic, modelling only this type of flows. TCP traffic, indeed, is more convenient than UDP for the identification of flows.

The first 3 packets of a TCP connection regard the 3-way handshake that can be recognized by looking at the SYN flag in the first packet (the request of connection), the SYN-ACK in the second (the reply from the request of connection) and a simple ACK in the third (an acknowledgment of the session opening), these last packets, eventually, can already include data.

From the reception of SYN and SYN-ACK we can define the client-to-server (CTS) and the server-to-client (STC) direction. We assign CTS to the direction from which the SYN is received and STC to the SYN-ACK.

Also the connection closing phase is simpler to identify. The correct connection closing procedure starts with a packet with the FIN flag set; the other host, than, will acknowledge the received FIN using an ACK packet, this procedure will only close half of the connection. Another way to close a connection is using the RST flag (in this case ACK is not needed). Sometimes it can happen that, for some reasons, an host stops sending packets without properly closing the connection. In this case we need to consider a timeout system to manage this situation and consider as finished a flow after the timeout expiration.

Regarding the filtering phase, with the proposed solution, we allow different thresholds for the two flow directions to have maximum flexibility (for example to be able to filter only in one direction). The gain in having separate thresholds will be tested in the performance evaluation phase, to check whether this number can influence the DPI accuracy performance.

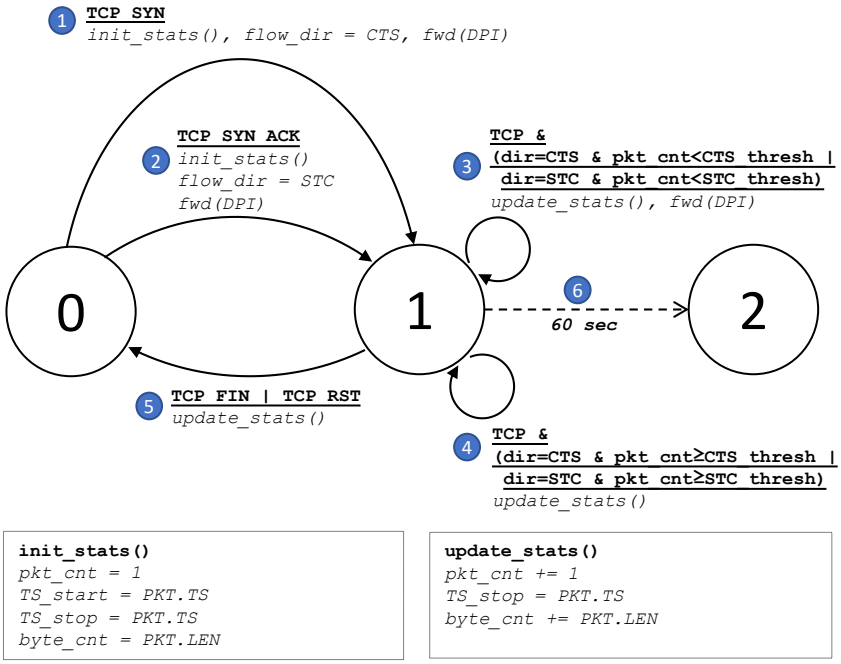

Figure 3.1: OPP State Machine for TCP traffic

As we can see in Figure 3.1 the TCP filtering procedure is modeled as a state machine. The main components of a state machine are states and transitions and hereafter we will detail the role of each of these:

- ***State 0*** represents the *default* state applied to any new flow (i.e.: a flow for which no packets have been received so far) or a completed flow (i.e.: a flow for which the FIN or RST packet has been received).
- ***State 1*** is the main state of our machine and represents an ongoing flow.
- ***State 2*** represents the so called *"To Be Collected"* state: a flow will go in this state only in case of timeouts (in our case 60 seconds). A timeout can expire when, for some reason, no packets of a flow traversed the switch for at least the duration of our timeout.

Transitions between states and related actions are illustrated hereafter:

- ***Transition 1***: This transition represents the connection opening in the

CTS direction.

The matching is performed on TCP SYN flag, and therefore with this transition the flow will also be initialized. The initialization comprises setting the related flow variables including a packet counter, flow-start and flow-end timestamps, and a flow volume (bytes) counter. The flow direction will also be set to CTS and the packet will be forwarded to the DPI.

- ***Transition 2***: This transition represents the connection opening in the STC direction.

  The matching is performed on TCP SYN and ACK flags, and therefore with this transition the flow will also be initialized. The initialization comprises setting the related flow variables including a packet counter, flow-start and flow-end timestamps, and a flow volume (bytes) counter. The flow direction will also be set to STC and the packet will be forwarded to the DPI, similarly to what is done in Transition 1.

- ***Transition 3***: Auto transition, this transition does not perform a state change, it represents the phase in which the switch will forward packets to the DPI.

  In this transition we have two separate cases merged in one with an OR condition. The first regards the CTS direction. In this case we will check if the packet counter is smaller than the CTS threshold. The second condition regards the STC direction. In this case we will check if the packets counter is smaller than the STC threshold. In both the situations, the following actions will be performed: (1) the packet counter will be increased by 1, (2) the end timestamp will be updated to the current instant, (3) the byte counter will be increased based on the packet length, and finally (4) the switch will forward the packet to the DPI.

- ***Transition 4***: Auto transition, this transition does not perform a state change, it represents the phase in which the switch will no more forward

packets to the DPI because the packet counter reached the threshold.
In this transition we have, as the previous one, two separate cases merged in one with an OR condition. The two conditions match on whether the direction of the packet is CTS or STC, and, respectively, they will check if the packet counter is higher or equal to the CTS or STC threshold. In both these situations, the following actions will be performed: (1) the packet counter will be increased by 1, (2) the end timestamp will be updated to the current instant, (3) the byte counter will be increased based on the packet length. However, in this transition, the switch will not forward the packet to the DPI.

- ***Transition 5***: End of connection.
  This transition will be performed on the arrival of a packet with FIN or a RST flag set. In this case packet counter, end timestamp and byte counter will be updated, the packet will not be forwarded to the DPI (also if the packet counter is smaller than the threshold).

- ***Transition 6***: This transition happens when the timeout expires.
  Every time a packet is received, the corresponding flow timeout is reset. A timeout expires when the two endpoints do not exchange any packet throughout the whole duration of the timeout. It will be performed only a state change (from State 1 to State 2), no update on data variables will be done.

The state machine fits perfectly our TCP connection model, it allows us to represent the beginning of the connection and the filtering phase. It also allows us to represent the statistics collection phase performed also after the first forwarding phase. Moreover, the OPP's packet-based state transitions are very useful for our application.

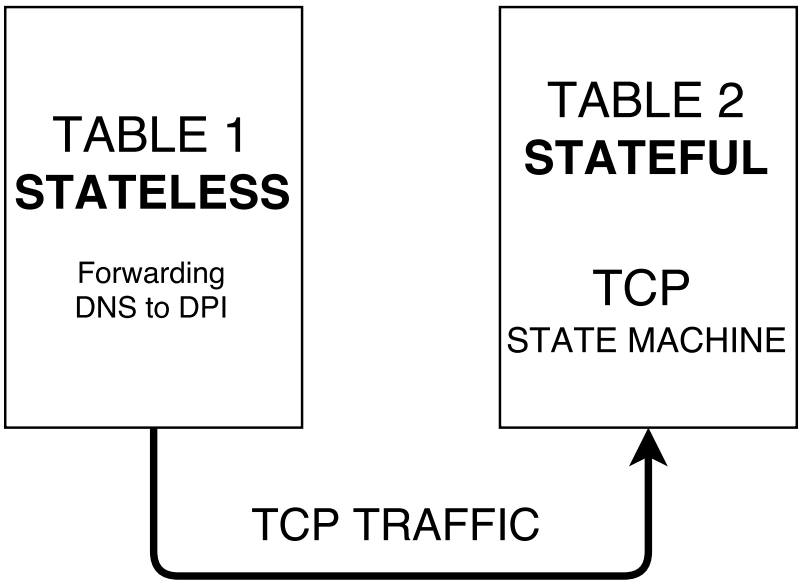

Figure 3.2: The Switch pipeline

3.2.1 Implementation

3.2.1.1 Pipeline

Talking about the switch itself, it is configured with a two stage pipeline as shown in Figure 3.2. The first stage is a classic OpenFlow stateless table. This table is used to install rules regarding the normal forwarding functionality of the switch, it also redirects the TCP packets to the second table. This table also contains a stateless UDP rule used only to redirect DNS replies to the DPI. We observed that this traffic is fundamental to make correct and high quality classifications. The second stage is a stateful OPP table: it implements everything described before and maintains track of the TCP flows and their statistics.

The combination of the two stages gives maximum flexibility, the stateful one can optimize the DPI work making flow classification efficient, the stateless instead can be exploited to perform application-aware tasks like traffic shaping or others (see Section 2.2.4) using the knowledge about traffic extracted through classification.

3.2.1.2 From State Machine to OPP

In this section we will translate the State Machine described in Section 3.2 into an OPP-compatible EFSM table. In order to do that, we are also going to define the Lookup and Update Scopes, the global and flow data variables and the conditions table.

Since we are dealing with TCP traffic, our Flow Key it is exactly the TCP 5-tuple. When a packet arrives in a stateful stage, a Flow Key is extracted (in our case the 5-tuple) and used to perform a flow context lookup from which flow register values and the current flow state are extracted, and afterwards the condition evaluation phase is executed. Then, a match on the EFSM table is performed, based on the outcome of the condition evaluation, the state and the packet header. The result of the match is a triplet comprising the next state, update actions and packet actions.

Lastly, packet actions are performed, Update Flow Key is extracted (in our case the Update Flow Key is the same as Lookup) and update actions (related to flow statistics collection only) are performed.

Priority	C0	C1	C2	State	Packet Fields	Next State	Packet Actions	Update Actions
HIGH	*	*	*	1	TCP RST=1	0		R0++ R2=pkt.ts R3+=pkt.size
HIGH	*	*	*	1	TCP FIN=1	0		R0++ R2=pkt.ts R3+=pkt.size
HIGH	*	*	*	0	TCP SYN=1 ACK=1	1	DPI()	R0=1 R1=pkt.ts R2=pkt.ts R3=pkt.size R4=0
LOW	*	*	*	0	TCP SYN=1	1	DPI()	R0=1 R1=pkt.ts R2=pkt.ts R3=pkt.size R4=1
LOW	0	*	1	1	TCP		DPI()	R0++ R2=pkt.ts R3+=pkt.size
LOW	*	0	0	1	TCP		DPI()	R0++ R2=pkt.ts R3+=pkt.size
LOW	1	*	1	1	TCP			R0++ R2=pkt.ts R3+=pkt.size
LOW	*	1	0	1	TCP			R0++ R2=pkt.ts R3+=pkt.size

Table 3.1: Extended Finite State Machine table for TCP traffic

Table 3.1 illustrates the EFSM table. This table is the main part of the implementation, it represents the low-level details of the stateful forwarding that we described before in the State Machine in Figure 3.1. C0, C1 and C2 are condition fields, presented later on in this section, while the state field denotes the current flow state as in Figure 3.1. In this tabular version, we see more transitions than the State Machine because there is not a compact way, in OpenFlow (and in OPP as well), to define the *OR* condition in the matching fields. This type of conditions are mapped to 2 different rules. The table clarifies that the matching will be performed not only on the packet header fields (like in standard OpenFlow), but also on the conditions (C0, C1 and C2), as well as on the current state of the flow. Priority in the table is used only to be sure that matching is performed in the correct way in case of overlapping rules. For example, it ensures that when the switch receives a RST or FIN flag, it will perform the desired flow closing actions as well as a

transition to State 0, instead of the (lower priority) rules from 5 to 8.

The timeout based state transition is handled through a timeout and an idle rollback state which can be set for every flow. For our implementation we set an idle timeout of 60 seconds and an idle rollback state 2 meaning that if no packets is received within 60 seconds for a given flow, that flow will be transitioned to State 2 marking it as expired.

Variable	**Meaning**
R0	Counter of the number of packets
R1	Start timestamp
R2	End timestamp
R3	Bytes transferred
R4	Direction

Table 3.2: Flow Data Variables

Variable	**Value**	**Meaning**
G0	*Programmable*	CTS Threshold of packets to forward at the DPI
G1	*Programmable*	STC Threshold of packets to forward at the DPI
G2	0	Placeholder to know the flow direction

Table 3.3: Global Data Variables

In Table 3.2 and Table 3.3 we can find the list of all the memory registers (Section 2.2.3.1). These registers can be differentiated in *Flow Data Variable* and *Global Data Variable*, the first are local to each flow, the second are global, shared by all the flows.

The Flow Data Variables are used to store the flow related statistics, they are updated by the actions specified on the EFSM table and will be gathered by the controller when the flow is expired.

Global Data Variables are used to store thresholds regarding the number of packets to forward at the DPI in both directions (CTS and STC) and a placeholder to identify the direction of the flow.

The Flow Context associated to each flow is thus composed by the state plus the five flow data variables. With the proposed solution we have, for each flow, two entries in the Flow Context Table. We divided the two direction as two independent Flow entries, to allow the definition of two different thresholds for the CTS and STC direction, increasing the flexibility of the solution.

Condition	**1^{st} Operand**	**2^{nd} Operand**	**Operator**	**Meaning**
C0	R0	G0	$\geq$	Used to check if the current packet has to be forwarded to the DPI, making comparison on the CTS threshold. C0=0: forward to DPI C0=1: do not forward to DPI
C1	R0	G1	$\geq$	Used to check if the current packet has to be forwarded to the DPI, making comparison on the STC threshold. C1=0: forward to DPI C1=1: do not forward to DPI
C2	R4	G2	$>$	Check if the current packet belongs to the CTS of the STC direction. C2=0: STC C2=1: CTS

Table 3.4: Condition Table

Table 3.4 describes the conditions used to check when a direction of a flow has

reached the packets threshold, and then it will no more be forwarded to the DPI. Condition 2 is used to check the direction of the flow, to make it possible to set direction-dependent thresholds.

3.2.1.3 Switch start-up

The switch will be programmed by the controller in the start-up phase (i.e. as soon as it is detected by the controller). The controller is going to set Lookup and Update Flow Key, Global Data Variables (their contents), EFSM table (all the transitions representing the State Machine), but it is not going to set any Flow Data Variables. These variables together with the Flow Context Table are going to be populated by the switch itself performing the transitions, actions and updates present in the EFSM table. After the programming phase, the controller is not going to perform any actions on the switch, regarding the stateful part, except statistics collection. This allows to obtain high performance and scalability with respect to a controller-dependent counting and filtering.

3.2.1.4 Statistics Gathering

The controller needs to gather the statistics collected from the switch. The statistics are necessary only when the flow is expired, that is when the switch detects a FIN or a RST or when the flow expires for a timeout. There are two ways the controller can collect these statistics. The first is a direct notification from the switch (push-based): the switch notifies the controller when a flow changes state. The controller will receive notifications when a flow goes from State 0 to State 1 and vice-versa. Our interest is only on the notification of the transition from State 1 to State 0 (that corresponds to the reception of FIN or RST packets).

A clarification needs to be made: notifications are sent only when there is a

state change not based on timeout expiration. When a flow ends for a timeout expiration, the switch changes the flow to State 2, but in this case it does not send any notification to the controller. To manage this second case of flow ending, the controller will poll periodically the switch, sending a state request exactly for flows in State 2 (pull-based method for statistics gathering). With this last request, the controller also triggers the flushing of these entry from the State Context Table.

In the current OPP implementation state notifications are synchronous to packet processing and state changes, thus push based statistics gathering can introduce little overhead on the switch packet processing. To reduce this overhead we can think about removing the notification system (push-based) method to collect statistics. We can modify our OPP implementation, disabling the notifications; by doing so the controller would not receive statistics from the switch but it would need to gather them directly through the pull-based method using state request statistics for State 2. Moreover, still in this case, we need to modify slightly our state machine and our EFSM table to change the *Transition 5*: with this implementation this should instead be towards State 2. Applying this change can reduce the overhead for both controller and switch; the switch would not need to send synchronous state change notification, and the controller could autonomously choose a polling rate as to maximize its own fast-flow statistics

3.2.1.5 *Bi-Flow* Optimization

The solution proposed can be further optimized, using a new functionality of OPP called *Bi-Flow*. With this functionality we can specify a double Update Flow Key: a flow can perform an action in the update phase in up to two flow entries with different Flow Key. This functionality can be easily exploited in the case of RST flag connection closing. In case of RST, with the current solution, we close only half of the connection, the other direction is left open

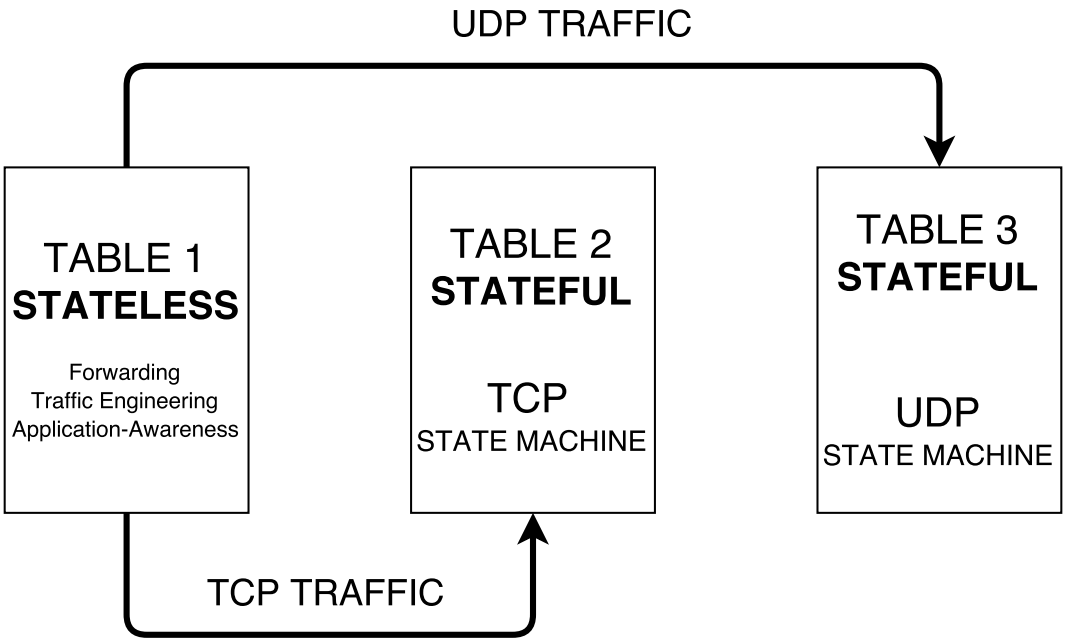

Figure 3.3: Extended switch pipeline

and will expire eventually with the timeout. With the Bi-Flow we can close both direction on the reaching of the RST flag, using our Update Flow Key with source and destination inverted. This optimization, however, is not crucial because we can expect that we will receive a small quantity of RST flags. For the sake of clarity we decided not to include in the state machine even if it has been implemented in our application.

3.2.2 Possible Extensions

3.2.2.1 UDP State Machine

The proposed solution and implementation can be easily extended to support UDP traffic. The pipeline can be extended with another stateful table (Figure 3.3), where we can program the Finite State Machine for the UDP traffic. There will be two tables: one stateful table for TCP flows, and another one for UDP flows. TCP packets will be forwarded to the first table, whereas UDP packets will instead go on the second one. The UDP FSM (Figure 3.4) will

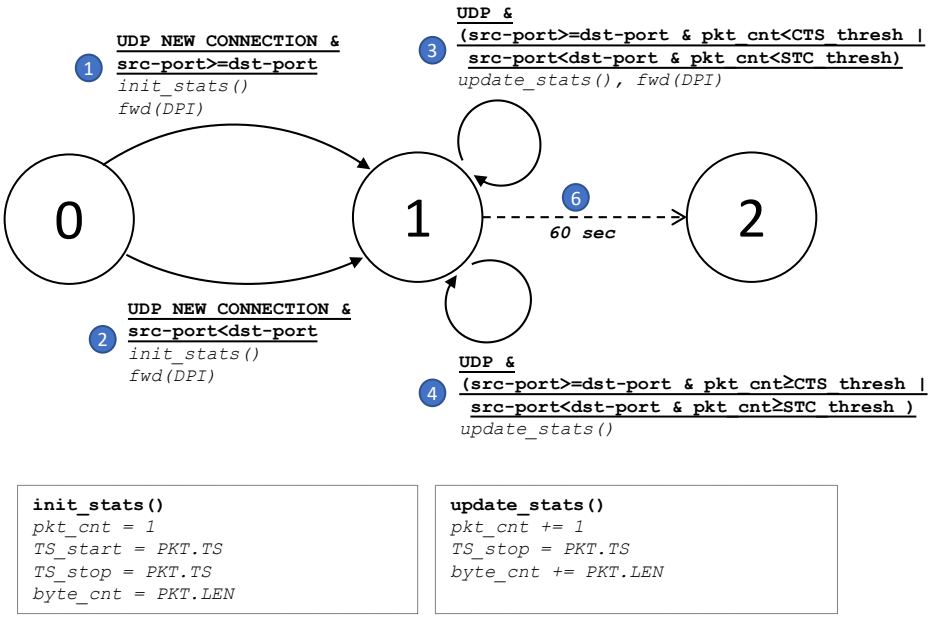

Figure 3.4: UDP traffic finite state machine

be different from the TCP counterpart, mainly because we would not have the TCP flags on which we can perform the matching to detect the connection starts and ends phases.

The starting point of a UDP connection is considered the time when we first see a packets belonging to a flow. The definition of flows is the 5-Tuple (the same as in TCP but with UDP source and destination ports). Also with UDP we can make the differentiation between the two direction of a flow: in this case, due to the lack of flags, we need to define a direction in some way. In this example we defined as the CTS direction the one in which the UDP source port is greater or equal to the UDP destination port (we suppose that the server will have one of the well known port) and STC as the opposite. Another drawback of the lack of flags is related to the closing of the connection. In this case we need to rely completely on the timeout procedure: every flow will end when the timeout expires. This implies that the controller will never receive the notification for the state change, and therefore with UDP traffic the statistics need to be gathered directly by the controller with the polling

procedure described above.

Priority	C0	C1	C3	State	Packet Fields	Next State	Packet Actions	Update Actions
HIGH	0	*	1	1	UDP		DPI()	R0++ R2=pkt.ts R3+=pkt.size
HIGH	*	0	0	1	UDP		DPI()	R0++ R2=pkt.ts R3+=pkt.size
HIGH	1	*	1	1	UDP			R0++ R2=pkt.ts R3+=pkt.size
HIGH	*	1	0	1	UDP			R0++ R2=pkt.ts R3+=pkt.size
LOW	*	*	1	0	UDP	1	DPI()	R0=1 R1=pkt.ts R2=pkt.ts R3=pkt.size
LOW	*	*	0	0	UDP	1	DPI()	R0=1 R1=pkt.ts R2=pkt.ts R3=pkt.size

Table 3.5: EFSM table for UDP traffic

Condition	1^{st} Operand	2^{nd} Operand	Operator	Meaning
C3	UDP src port	UDP dst port	$\geq$	Check if the current packet belongs to the CTS or the STC direction. C3=0: STC direction C3=1: CTS direction

Table 3.6: UDP Extension for Condition Table

The mapping of the EFSM to OPP low level rules is similar to the one of TCP. In Table 3.5 we can find the tabular representation of the FSM for UDP traffic. Note that, compared to Figure 3.1, no matching is performed on flags, furthermore there is no transition back to the State 0. We add also the condition C3 to detect the flow direction (described in Table 3.6), the others conditions and registers are the same as in TCP, they can be found in Table 3.4 and Table 3.2. For the sake of simplicity this manuscript will only present the results obtained for TCP flows.

3.2.2.2 DPI Feedback

Another extension that can be made is to implement a feedback scheme where the DPI can send a packet to signal the end of the classification, as done in [15]. With this optimization the switch can block the forwarding of packets to the DPI in advance, further reducing the number of packets analyzed by the DPI. We can trivially extend our EFSM to include this optimization, however we do not believe this would bring significant added gain.

3.2.2.3 Stateful Application Awareness

We can also think of extending our proposal to perform application-aware tasks directly on stateful table. To do that we would then define a sixth flow data variable that identifies the category of the flow, which would be set by the controller upon receiving the classification result from the DPI itself. Doing that we would be able to define forwarding actions based on the category of traffic (i.e. prioritize VoIP application, slow down P2P or block malicious traffic). This approach is more optimized that the usage of the stateless table because we do not need to define an entry for each flow with the respective actions. With the stateful only solution we predefine the categories and then the new flow data variable distinguishes between them. Moreover, the controller can also redefine the state machine if we need to introduce other traffic categories in real-time.

3.2.3 Hardware Feasibility

The authors of OPP demonstrated that their architecture is implementable in hardware. In this section we would like to evaluate the memory requirements of our application.

Considering the version with TCP only, we would require 8 EFSM table flow

entries, whereas the one including also UDP needs 14 flow entries. Regarding the number of state entries required, this value depend directly the number of flows present on the traffic. The FPGA based prototype proposed in [21], supports no more than 128 EFSM table entries in TCAM (more than enough for our application) and up to 4000 entries on the flow context table in RAM. These value allows to support less than 2000 TCP connections at a time (some flow entry will be in the State 2 waiting to be collected from the controller and some other on State 0 waiting to be flushed by the switch). This limiting value can be overcome with an ASIC based implementation: in this case the flow context table (according to [21]) can support up to 256K EFSM table entries and the flow context table can contains around 1 million of entries. These values can even not be enough in scenarios with high-capacity high-speed links. The main limitation is due to the fact that the read and write on RAM, within the actual architecture, must be performed at clock chip speed so we cannot use slow DRAM (which require more than 1 clock cycle at 1 GHz to read and write), but we must use the fast SRAM (which as drawbacks cannot be as big as DRAM and it is more expensive). There is some work, like [27] in which the authors claim that we can relax these constraints allowing to have more cycle to execute more complex operations directly on data plane or, for example, access to a slower and bigger memory. The possibility to use DRAM, larger and cheaper than SRAM, allows us to remove the above mentioned limitations making our application deployable also on high speed, high capacity link with more than 1 Million TCP connections alive at the same time.

3.3 Use cases

The solution proposed can fit best in special networking environment, for example enterprise or campus networks. In this type of networks there can be

particular configurations that allows packets of the same flow to go in different part of the network, or there can be internal cache servers used to speed up navigation. Deploying a DPI in these type of networks poses great challenges: DPIs need to inspect all the traffic so they must be placed in a point of network where all traffic will pass. With our solution and in general with an SDN-based solution, we can exploit the programmability of the network to deal with such an environment. To make an example we will present a common problem found in some network (for example in Politecnico di Milano campus network).

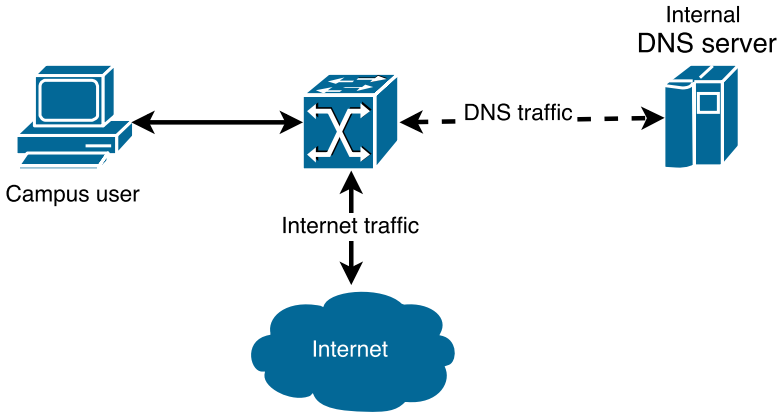

Figure 3.5: Use Case: Internal DNS Server

Lots of networks has an internal DNS server. It is used to greatly reduce the DNS traffic directed to the outside, speed up Name Resolutions and, at the same time, reduce Internet traffic. Deploying a DPI in one such an environment it would have been problematic. However, with the solution proposed in this work, this problem is not anymore an issue. We can place smartness (OPP switch) in strategic points of the network, in a distributed way and from these points forward packets to a central unit where the DPI is placed.

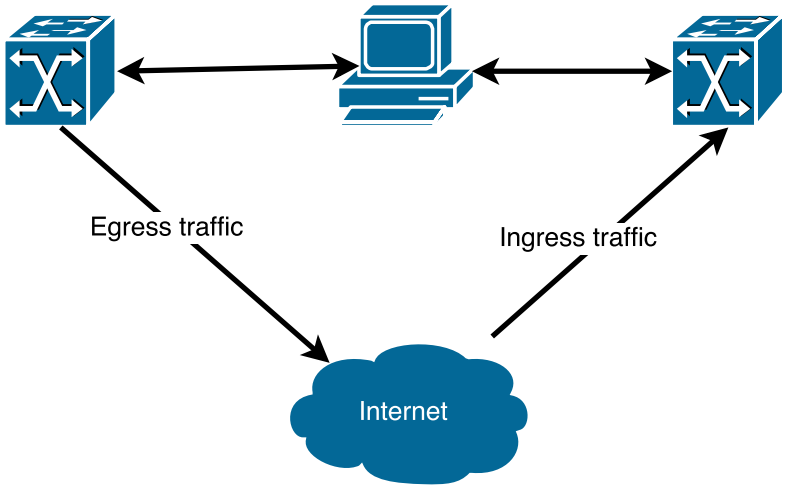

Figure 3.6: Use Case: different Egress and Ingress point

Another use case can be a network with different ingress and egress point. We can have a network configuration that allows traffic in one direction to exit the network from a specific point and traffic in the other direction to enter the network from another point. This case is difficult to manage with common DPI deployments. With this work's solution the problem is no more present. We place two OPP switches in the two points and we can make filtering and forward packets to the DPI from the two locations

From these use cases we can see how this solution can adapt to several network configurations allowing to deploy DPI in networks where it would have been impossible, or at least would have required a complete network redesign. Whereas, with our solution we need only to replace some networking elements and then the DPI can be deployed with no other effort.

Chapter 4

Performance Evaluation

In this chapter we are going to present the numerical analysis conducted on the testing environment. We implemented our SDN application using the switch and the controller developed within the *BEBA project* [28]. The switch [29] is an extension of CPqD OpenFlow 1.3 software switch [30] with the support of OPP stateful processing. As controller we used [31], a modified version of the the widely adopted OpenFlow controller Ryu [32] extended to support OPP protocol. Everything was tested in the network emulator Mininet [33, 34] using a pre-built Ubuntu virtual machine.

The reference testing environment (Figure 4.1) is composed of 2 hosts connected through a software switch. One of the ports of the switch is connected to the Internet through a NAT node. Also the DPI is directly connected to the switch even if with our solution nothing prevent us to place it within the same machine hosting the controller.

Regarding the DPI, we decided to use nDPI [14], an Open Source high-speed DPI which is provided as a Software Development Kit (SDK). We based our implementation of the DPI on the example program provided with the SDK to make our evaluations.

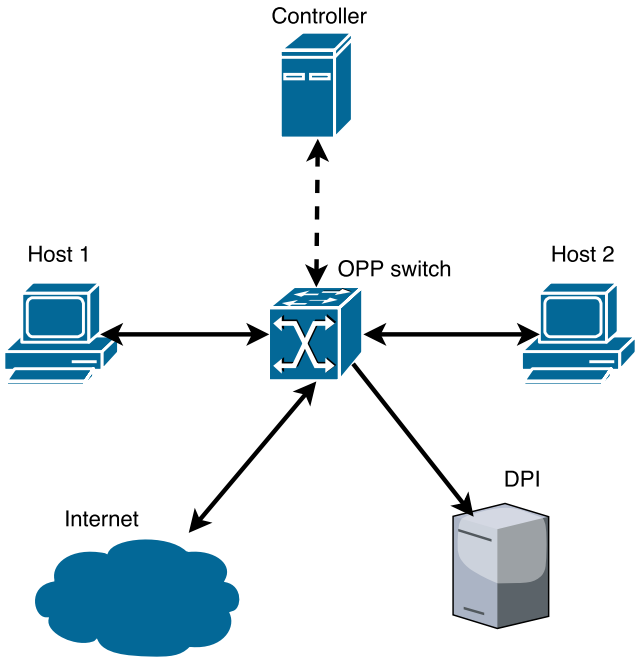

Figure 4.1: Testing environment topology

The application code is available as open-source software at [35].

4.1 Description of the tests

The following analysis have been conducted:

- ***Classification Accuracy***: evaluation of the implications of our filtering scheme on the classification accuracy (Section 4.2);
- ***Filtering Impact***: evaluation of the impact of filtering on the traffic analyzed by the DPI in terms of number of packets and quantity of bytes (Section 4.3);
- ***Software Switch Performance***: evaluation of the impact of filtering and statistics collection on the switch performance (Section 4.4);
- ***DPI performance***: evaluations of performance of the DPI analyzing filtered traffic (Section 4.5).

We finally evaluate advantages and disadvantages of our proposed solution in Section 4.6 and we proposed a *reactive* solution based on Standard OpenFlow in Section 4.7.

4.2 Classification Accuracy

This section presents the effects of the traffic filtering on the classification accuracy. First, we need to define our ground truth. As in [13], we classified the traces without making any sort of filtering and used the output of the DPI obtained with such configuration as the reference classification outcome. This output contains all the flows that the DPI is able to classify.

As already described, for the sake of simplicity, we just implemented the EFSM machine capable to filter TCP traffic only. Therefore, for this reason, before running the classification accuracy analysis, we filtered the traces leaving only TCP and DNS traffic. While our traces contained a subset of UDP flows (for P2P, VoIP, and QUIC traffic [36]), most of the traffic was still TCP-based.

The classification accuracy is defined as the percentage of flows correctly classified by the DPI when filtering the first n-th packets for each flow, with respect to the ground truth obtained inspecting the trace entirely. An accuracy of 100% means that the DPI, analyzing the filtered traffic, is able to classify the flows in the same exact way as analyzing the complete trace.

For the sake of simplicity the traffic trace has been filtered by a Python program functionally equivalent to the switch programmed with our application. For this program we read directly the PCAP trace file using *dpkt* [37] library. Before doing any numerical evaluation we verified that the traces obtained from the filtering performed by the switch and by the Python program were equivalent: we generated Internet traffic from the Host 1 of Figure 4.1, we recorded this traffic with *tcpdump* and, at the same time, we recorded the

filtered traffic between the switch and the DPI. After that, we filtered the complete trace with the Python program and verified that the obtained output contains the same packets of the trace captured between the switch and the DPI.

In this experimental evaluation we used 2 traces. The description of the cleaned traces (that is, those with TCP and DNS traffic only) are the following:

- **Trace 1**: more than 5 hours of captured traffic, with 3 Personal Computers and 2 smartphones connected. It contains normal Internet domestic traffic, and audio and video streaming.
  The trace size is 3.6 GB with more than 4 millions packets.
  nDPI identified 6886 flows.

- **Trace 2**: about 12 hours of captured traffic, the trace contains the same type of traffic and the same type and number of device as the previous one.
  In this case the trace size is 4.2 GB with about 4,4 millions packets.
  nDPI identified 10191 flows.

The DPI classification program reads the two traces (the original and the filtered one) from the PCAP files using *libpcap* [38]; it produces as output the classification results on a JSON file. This file has been analyzed by a Python program which parsed the output with the filtered trace and compared it with the result obtained with the ground truth to compute the accuracy.

The flexibility of our application allowed us to test the classification accuracy in three scenarios:

- **BOTH**: the DPI receives a variable number of packets from the two directions (the same number for both);

- **STC**: the DPI receives a variable number of packets sent from the server plus the first packet of the reverse direction;

- **CTS**: the DPI receives a variable number of packets sent from the client plus the first packet of the reverse direction.

The results obtained are shown in Figure 4.2 and Figure 4.4.

The number of packets on the X-axis of the graphs represents the total amount of packets per flow which go to the DPI (for example, 20 in the BOTH scenario means that 10 packets per direction go to the DPI).

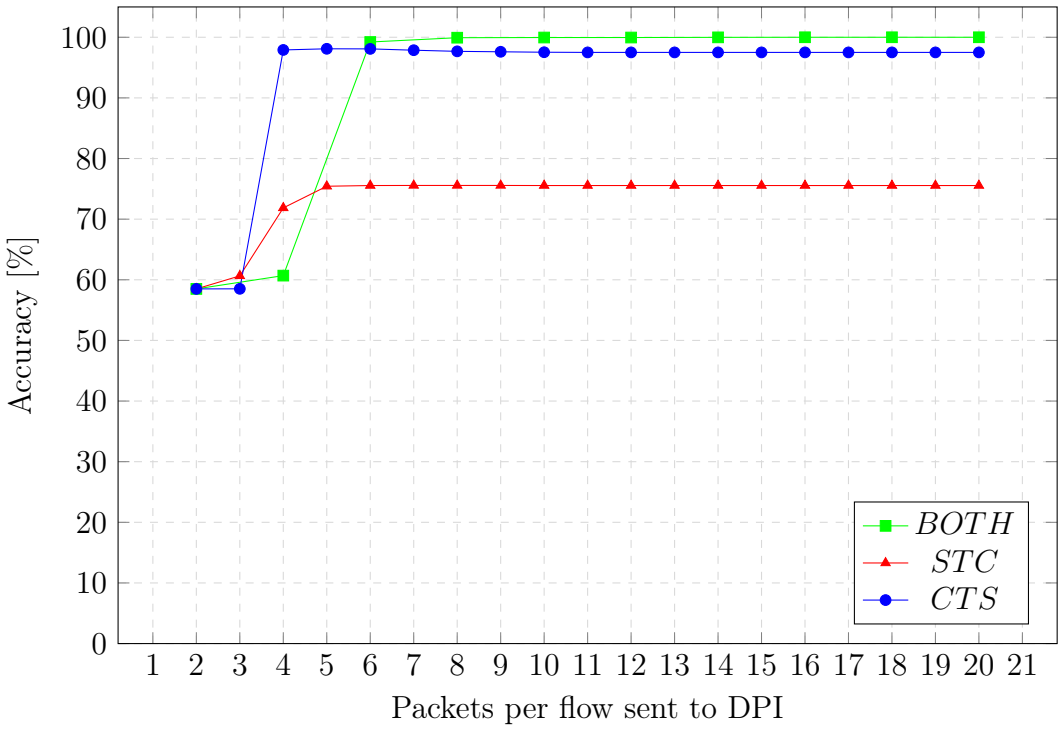

Figure 4.2: Classification accuracy: Trace 1 with nDPI

Figure 4.2 shows the classification accuracy obtained with nDPI on Trace 1. In this case the classification obtained by filtering only one direction of the traffic is lower than the one obtained by filtering both the directions with the same number of total packets. This means that the DPI works better when it inspects both directions of the traffic. For example with 10 packets we reached, with the BOTH case (5 packets per direction), 99.95% accuracy compared to 75.54% in the STC case and 97.53% in the CTS one. Furthermore, with only 16 total packets, filtering both the directions (8 packets for direction) we

obtained 100% accuracy and with only 6 total packets (3 per direction) we reached 99.22% accuracy. Given the results, we may also think of taking only the CTS direction (we however reached 98% accuracy with only 6 packets in this direction), but in general filtering both direction always reaches better results, starting from a threshold of 6 total packets.

Comparing the CTS and STC direction, we can see that the DPI can extract more information from the CTS direction: in particular there is a big gap between 3 and 4 packets (respectively, threshold 2 and 3 in the CTS direction, given that on the other direction we send always one packet). When we send 3 packets on the CTS direction, the DPI can extract a lot of information from the last packet. Analyzing the filtered PCAP trace using Wireshark, we saw that this particular result is due to the presence of the ***Client Hello*** SSL message. This message contains a fundamental field, the ***Server Name*** which identifies the specific service we want to contact, it is mandatory because there might be more than one service on the same machine and the server needs to reply with the correct SSL certificate. This SSL field can be easily exploited to make classification since it identifies the specific application the user is contacting. To further verify this observation, we also analyzed the output of nDPI with 2 and 3 packets in the CTS direction. From these outputs we can clearly see that with a threshold of 2 packets, the vast majority of misclassified flows are SSL flows that nDPI has associated to some specific service/application (i.e.: Google, Amazon or Spotify) on the ground truth. As soon as we added one single more packet in the same direction, we saw that all these incorrectly classified SSL flows became correctly classified with the right application/service.

By looking at the Figure 4.2 accuracy decreases augmenting the number of packets (in the CTS and STC filtering directions). We investigated this behavior and we identified that some generic SSL traffic (as was identified in our ground truth) was classified differently as Skype traffic or (correctly) as SSL

varying the number of packets. This behavior could be further characterized with an analysis similar to the one performed in [12] where they evaluated how the classification migrates between classes as a consequence of filtering.

Given the previous observations, we tried to differentiate misclassified flows in two categories: filtered flows whose classification is completely different from the one obtained in the ground truth and filtered flows whose classification is "similar" to the ground truth. nDPI outputs the classification as a sort of *path* (i.e. *SSL.Google*), where each segment adds an information deeper and deeper in the protocol stack. According to our need we might trade higher accuracy for a shallow classification. If we just want to get the percentage of HTTPS traffic in our network, having a flow classified as SSL, when filtered, and as SSL.Google in the ground truth should not imply degradation in the accuracy. The results of this evaluation are shown in Figure 4.3. The accuracy that we can reach considering a relaxed classification (i.e. allowing a shallow classification) is higher, as expected. Note that the + case curves in Figure 4.3 represent the accuracy when considering exactly this case. Specifically for the case STC (Figure 4.3(b)) we have an improvement of around 7% reaching, with 10 total packets (9 in the STC direction), 82.89% accuracy instead of 75.54%. Analyzing directly the nDPI classification output we checked that this 7% is exactly the case we mentioned before: flows that were classified as SSL in the filtered trace, but with an "upper layer" specific application in the ground truth. Regarding the CTS direction (Figure 4.3(c)) we reached almost perfect classification with the relaxation of constraints, we reach 98.27% accuracy, with an increasing of around 0.77% with respect to the previous classification.

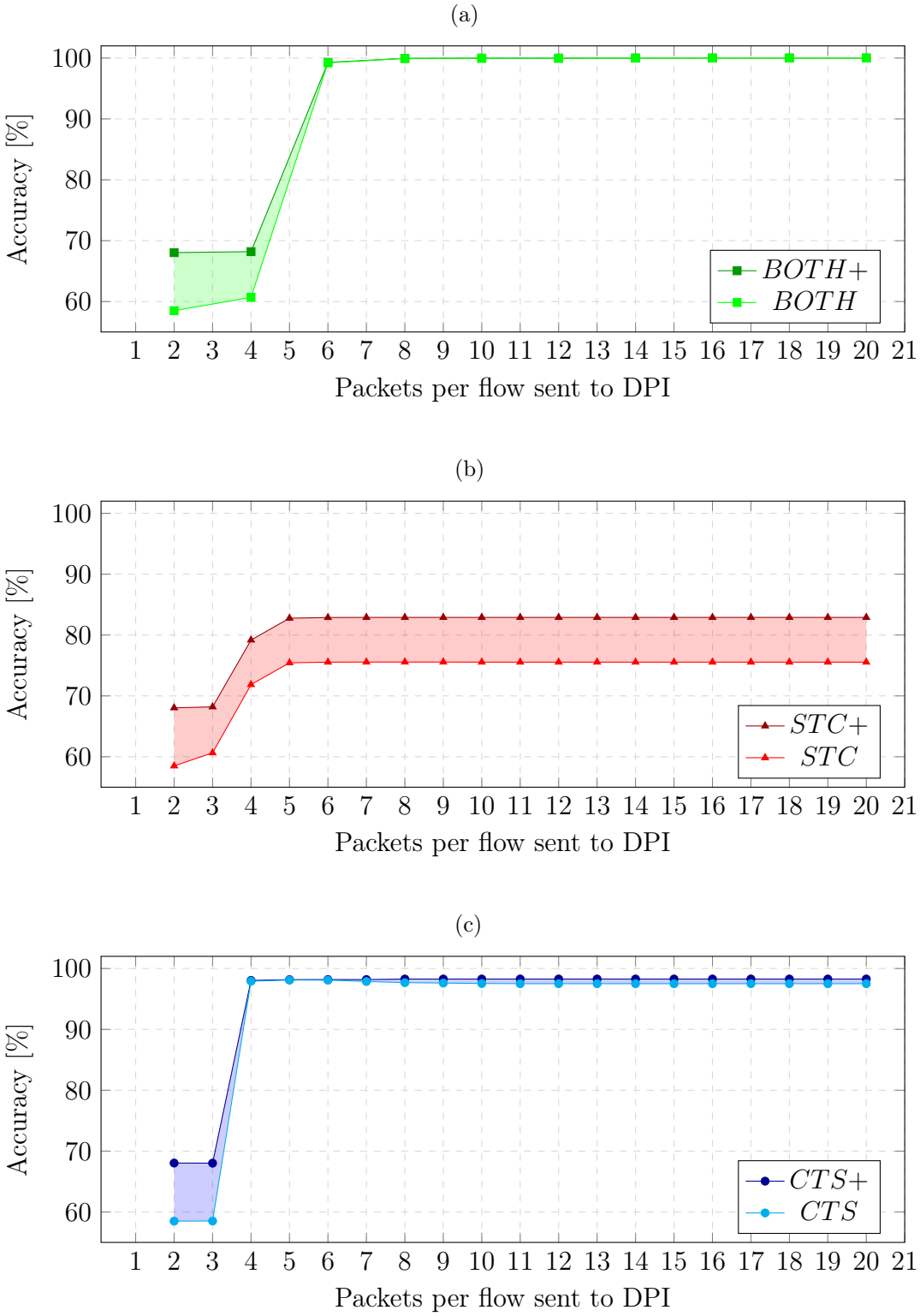

Figure 4.3: Comparison with shallow classification on Trace 1

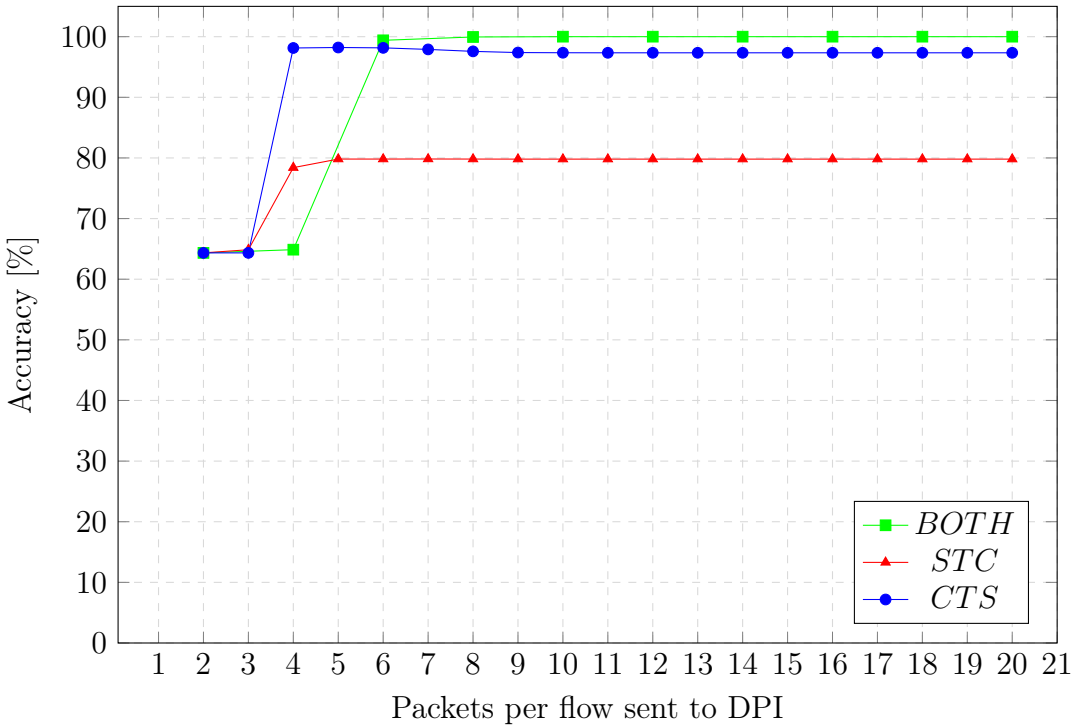

Figure 4.4: Classification accuracy: Trace 2 with nDPI

Figure 4.4 presents the results obtained with nDPI on Trace 2. The results are in line with Trace 1, we reached very similar values. Also with this trace we can see the big gap due to the SSL Client Hello. With this trace we reached 100% accuracy with only 10 total packets on the both case (5 packets per direction) compared to 79.81% with filtering on the STC direction and 97.35% on the CTS. These higher classification accuracy values are due primarily to the different type of traffic present in this trace.

Figure 4.5 is the same evaluation done for the Trace 1 relaxing the perfect match constraint. Also in this case we obtained a similar result as before. With Trace 2, for the case STC, we have an increase of 4.79% from 4 packets on, while on Trace 1 was around 7%. This value is different probably because in this trace there is less traffic in which nDPI can identify the specific application on top of SSL. The accuracy of DPI obviously is influenced also by the type of traffic that is present in the trace and on the capability of the DPI of identify a specific application on top of other protocols such as SSL.

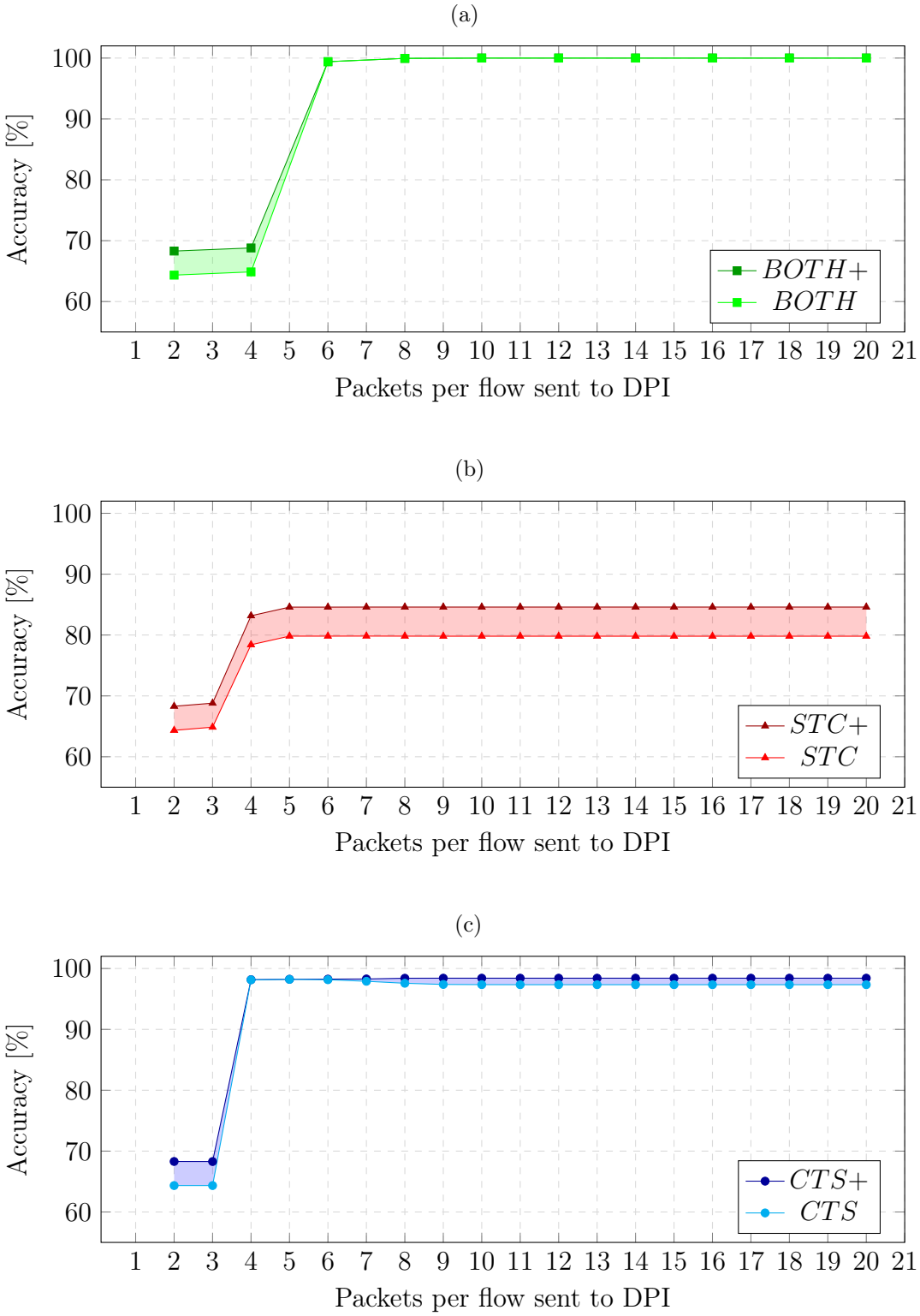

Figure 4.5: Comparison with shallow classification on Trace 2

From the results obtained we can state that analyzing only the first packets of TCP flows leads to a negligible loss of classification accuracy, confirming that the richest part of a TCP flow in terms of information that can be exploited by DPIs, is the first where application handshake is usually performed.

4.3 Filtering Impact

In this section we are going to analyze the effect of filtering on the number of packets and traffic volume. The evaluation is performed on the same two traces used in the previous section and also in a real trace from CAIDA [39].

4.3.1 Collected Traces

In this section we evaluated the filtering impact on the two traces used in the accuracy evaluation in Section 4.2.

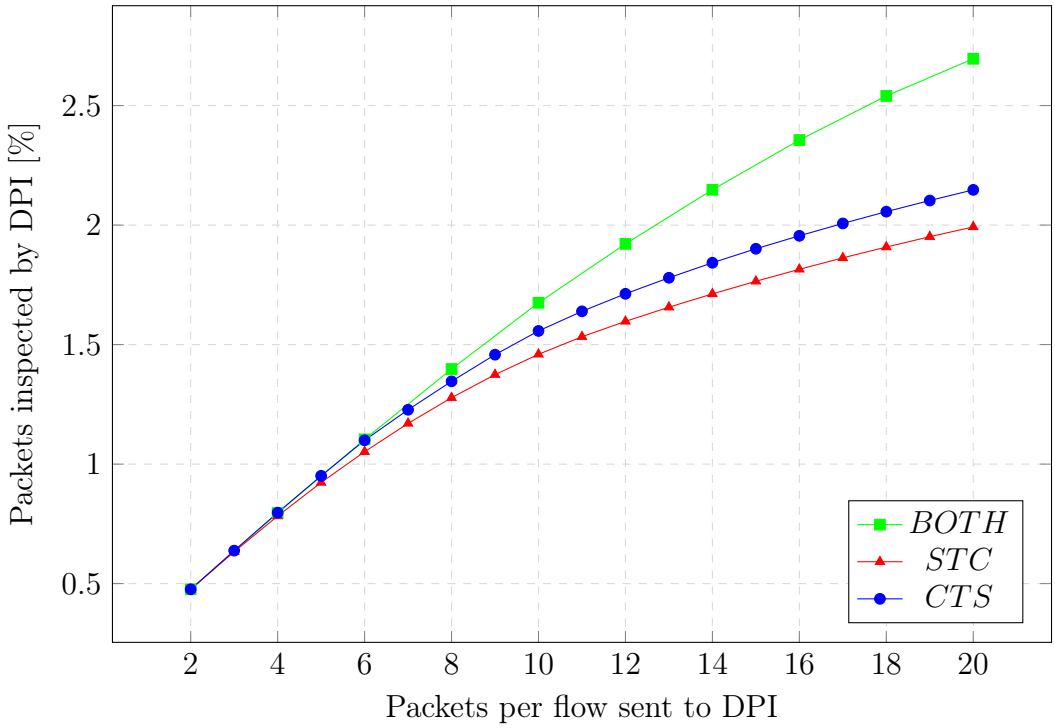

Figure 4.6: Percentage of packets to DPI on Trace 1

In Figure 4.6 is shown the percentage of packets inspected by the DPI varying the threshold. The legend is the same as the graphs discussed in Section 4.2. The gap between the CTS and STC filtering direction lines is due to different flows length and unbalanced number of packets in the 2 directions.

We reached 1.10% of packets in the BOTH case with 6 total packets (3 per direction) per flow. In absolute terms this value corresponds to a reduction of 2 orders of magnitude in the number of packets inspected by the DPI, passing from more than 4 Millions packets without filtering, to 44200 with this filtering threshold. With the value of 6 packets, in the evaluation done before, we reached 99% accuracy with nDPI. We can conclude that we have greatly reduced the number of packets maintaining an high level of classification accuracy.

The gain in filtering only one direction at a time is not so considerable to be taken in consideration. For example with a total number of 10 packets, with the BOTH case we reached 1.68% compared to 1.46% and 1.56% of STC and CTS respectively. Moreover, since these values are very small, they are not a discriminating factor on the choice of the three filtering cases, we better choose using the classification accuracy analysis done in Section 4.2.

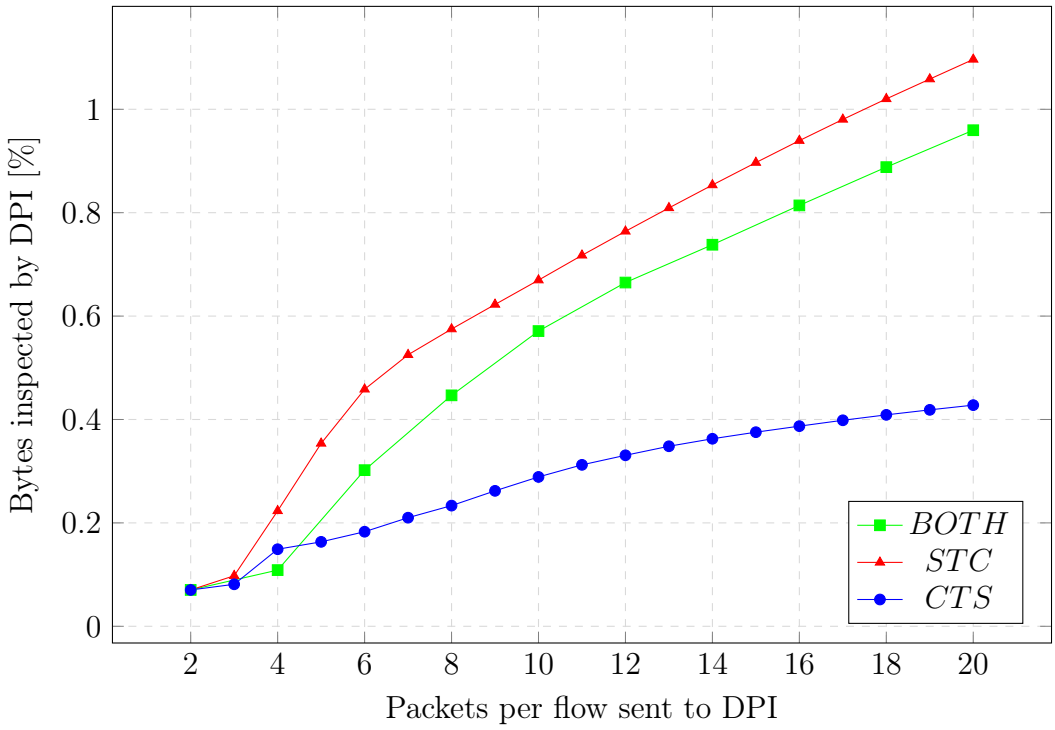

Figure 4.7: Percentage of bytes to DPI on Trace 1

Figure 4.7 is the plot of the percentage of bytes inspected by the DPI. As we can see the percentage is lower when we take into account the filtering performed only on the CTS direction. This is mainly due to the fact that since the traffic we are analyzing is mainly Internet traffic: in this direction we see primarily ACK packets and HTTP/HTTPS requests usually smaller than information transferring packets and HTTP/HTTPS response. The inverse holds for the STC direction, in this case the DPI has to analyze more traffic: in this direction web pages or video/audio streaming packets are transferred.

We reached only 0.3% of traffic volume, filtering 6 packets in the BOTH case (3 per direction). The first packets of a flow are usually small because they only transfer negotiation information. Also in this type of analysis we reached high classification accuracy with a very small quantity of traffic. Even in this case, the gain in filtering only one direction is not so high, we reached with a total of 10 packets for the BOTH case a value of 0.57% with respect to 0.67% for the STC case and 0.29% for the CTS. As above, since these values are

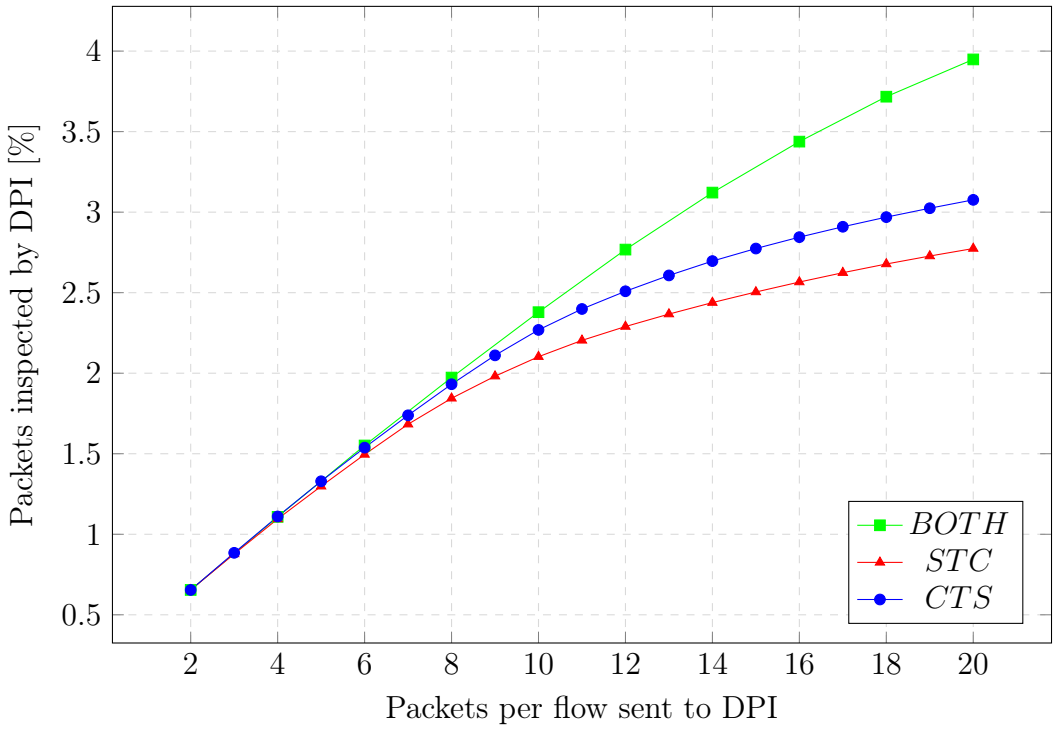

Figure 4.8: Percentage of packets to DPI on Trace 2

small, the choice of threshold should be done based on the analysis described in Section 4.2.

Figure 4.8 and Figure 4.9 refer to the Trace 2, the values obtained are slightly different from the ones obtained with Trace 1, this is due to the different traffic present in this trace as we illustrated also in the classification accuracy section. However, the difference is not so relevant, we reached only slightly different, but comparable, values.

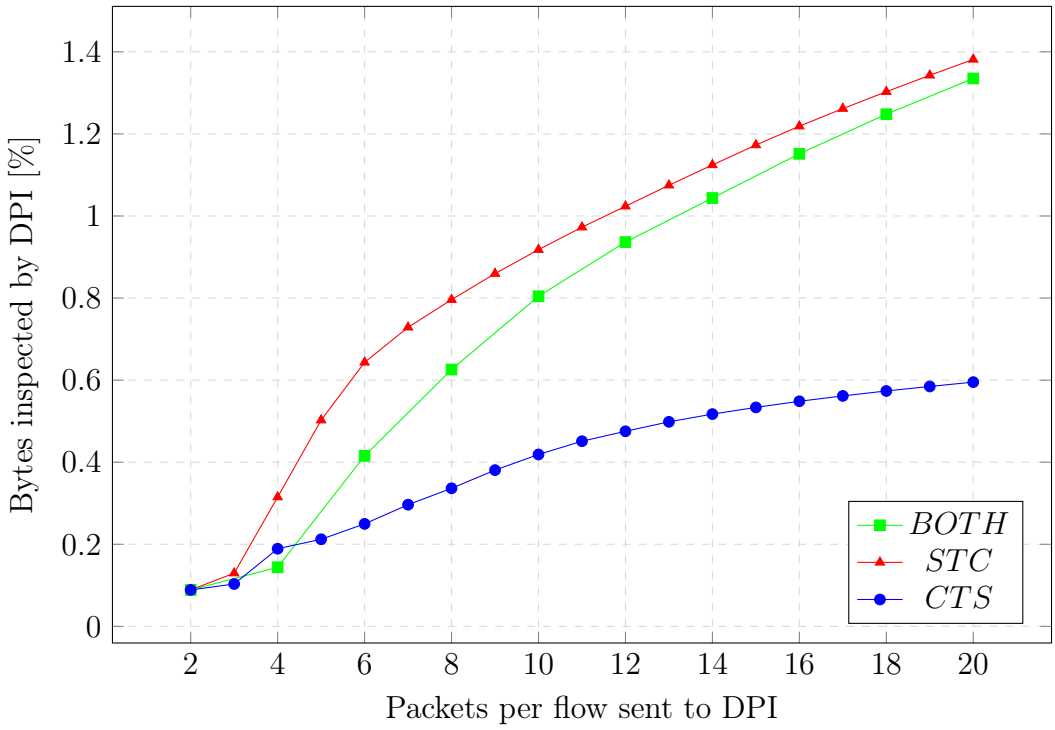

Figure 4.9: Percentage of bytes to DPI on Trace 2

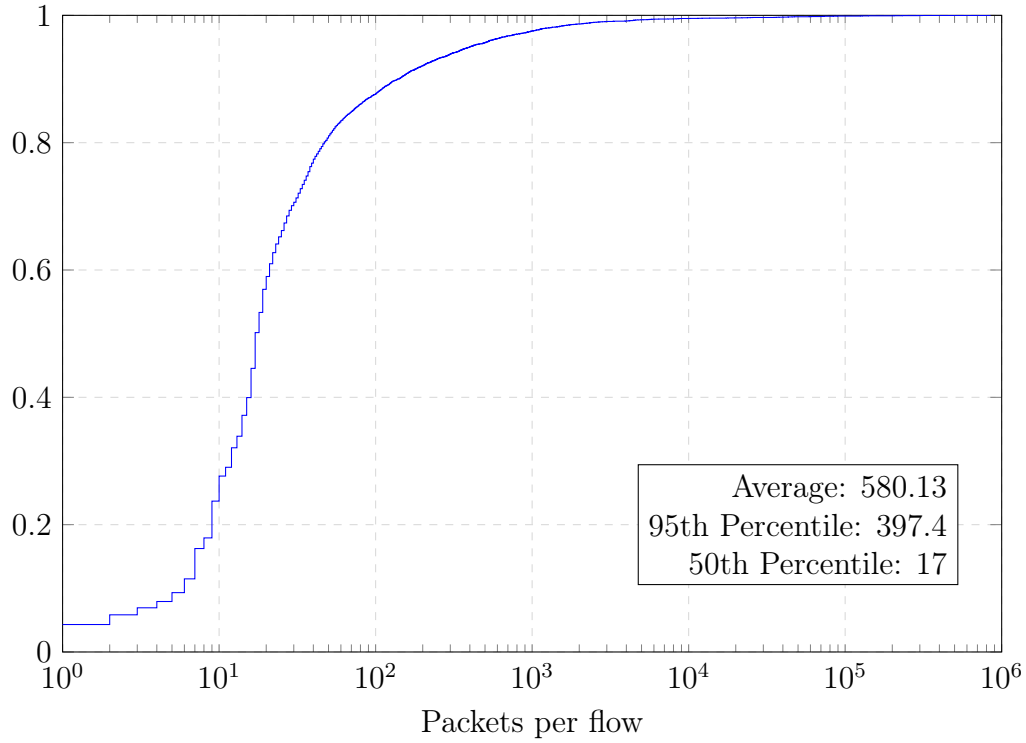

Figure 4.10: CDF of the packets per flow distribution of Trace 1

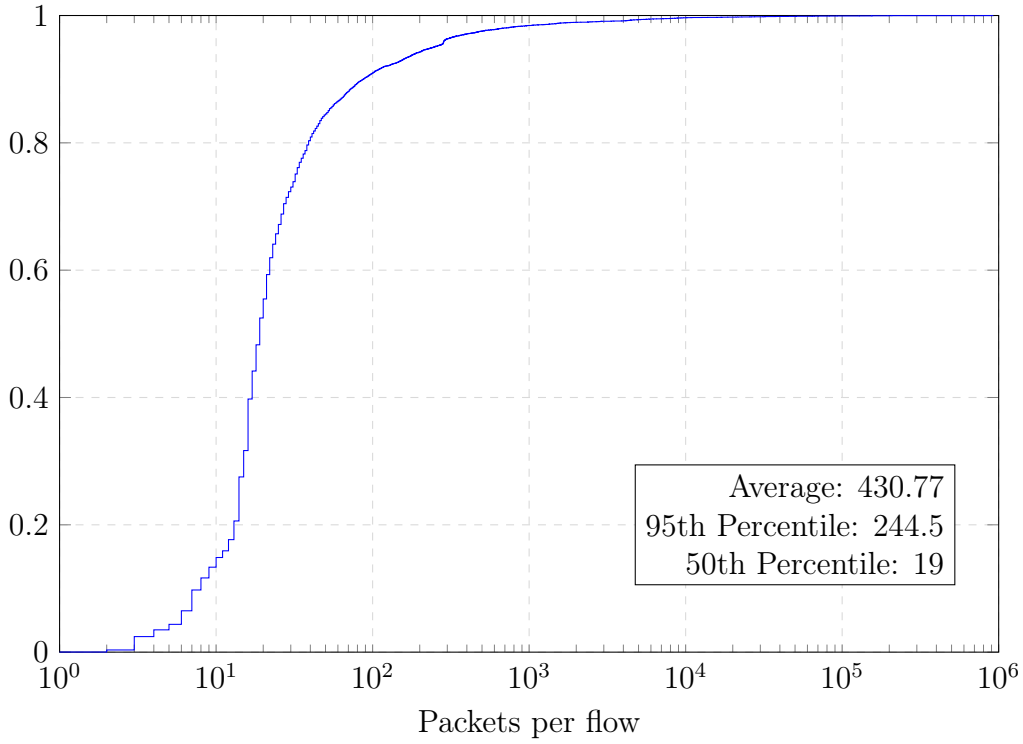

Figure 4.11: CDF of the packets per flow distribution of Trace 2

In the last two graphs, Figure 4.10 and Figure 4.11, we can see the packets per flow Cumulative Distribution Function (CDF). As expected, we observe a large number of small flows and a small number of very long flows. With our filtering technique we are going to gain most on the very long flows which are almost completely filtered out. Looking at the 50^{th} Percentile of both graphs we can see that this value is in line with our maximum used threshold. This means that 50% of flows are completely forwarded to the DPI and the other 50% are partially filtered. The first half of flows completely forwarded to the DPI are very small flows which represent a very small percentage of total traffic. From these graphs we can also see the different characteristics of the two traces, the two CDFs are slightly different, for example, on the CDF of the first trace (Figure 4.10) we can see that there is about 5% of flows with only 1 packet, instead in the second one (Figure 4.11) there is almost zero percent of such flows.

4.3.2 CAIDA trace

The trace used in this evaluation is a trace from CAIDA [39], captured on 19th February 2015 in Equinix Datacener in Chicago. The monitor point is connected to a 10GigE backbone link of a Tier1 ISP between Chicago, IL and Seattle, WA. This trace is anonymized using CryptoPan prefix-preserving anonymization and the payload is removed, making it impossible to analyze this trace with DPIs (although the IP length field is still present and therefore we can still evaluate the impact of filtering on the traffic volume). In this evaluation we have considered only the BOTH case for the threshold since is the most promising from the previously presented results in terms of classification accuracy.

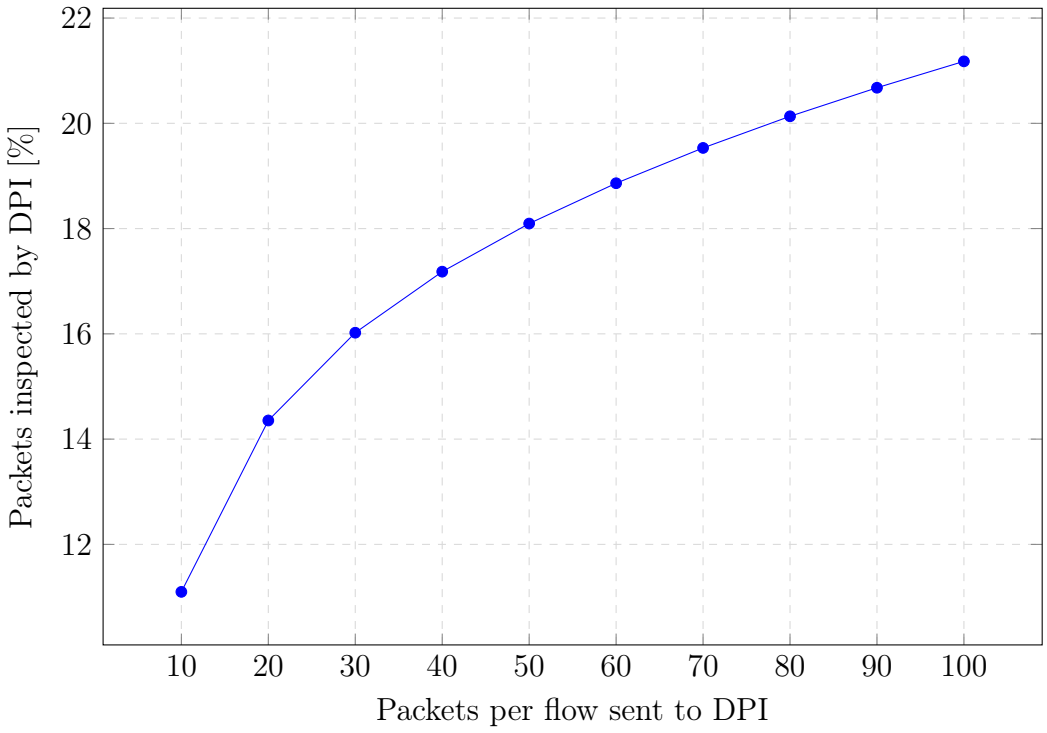

Figure 4.12: Percentage of packets to the DPI on CAIDA trace

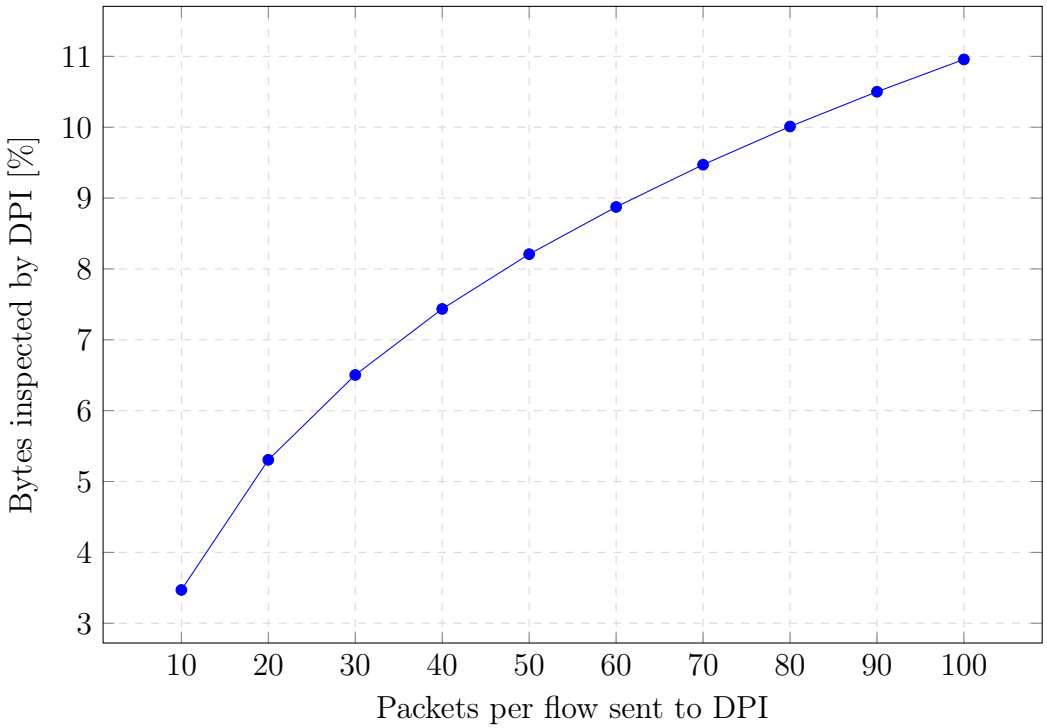

Figure 4.13: Percentage of bytes to DPI on CAIDA trace

Figure 4.12 and Figure 4.13 illustrate the filtering impact in a real traffic trace. We used different thresholds, larger than the maximum one used for the classification accuracy evaluation, to provide evidences that even rising the number of filtered packets, still significant offloading gains can be experienced. We can assume that the DPI can, however, reach high classification accuracy. As we can see, the filtering impact is huge also with large threshold. With a threshold of 10 packets we reached a reduction of more than 96% of bytes quantity that would have gone to the DPI and a reduction of more than 88% regarding the quantity of packets. With a threshold of 100 packets, we reached a reduction of more than 89% in terms of bytes and more than 78% in terms of packets. In conclusion, also with a quite large threshold (such as 100 packets) we can reach a huge reduction both in terms of bytes quantity and packets that must be inspected by the DPI.

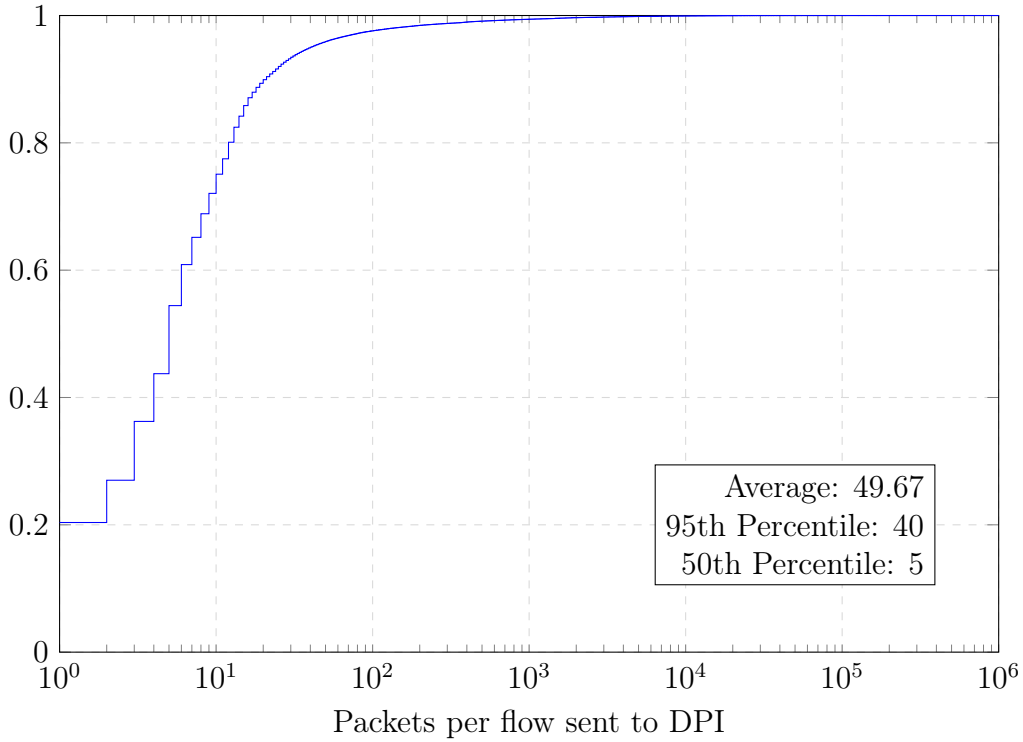

Figure 4.14: CDF of the packets per flow distribution of CAIDA trace

The values obtained with CAIDA trace are very different from those observed in the previous section (Section 4.3.1), for the reason that they were collected in two different network locations: CAIDA one is a core-network trace, whereas our traces were collected at the access network of a residential user. Indeed, by comparing the CDF of the packets per flow on the CAIDA trace in Figure 4.14 with the ones from our traces (Figure 4.10 and Figure 4.11), we have a completely different distribution so the traces are not similar. To make a real comparison we may want to analyze the percentage of saved traffic per flow. Having a distribution of this value, we can make a comparison and see how much we are going to save in terms of packets and bytes flow per flow.

Another observation that we can make regards the quantity of complete flows that are sent to the DPI varying the threshold. This value can be obtained through the CDF. We can see that sending 40 packets to the DPI correspond to sending completely about 95% of flows. However, this value brings to a saving

of 82% of packets to the DPI and 92% in terms of traffic volume. In absolute terms this means that on average the DPI has to process a throughput of 426 Mbps rather than 5.7 Gbps and a packets rate of 127 Kpps rather than 741 Kpps. This consideration can bring to the conclusion that 40 packets per flow is still a high threshold, and we expect that a very large fraction of the flows will be correctly classified while still ensuring a huge traffic reduction.

The contribution of performing this evaluation on CAIDA trace is to provide further evidences that even when collecting traces to a different network location, significant offloading benefits can still be experienced.

4.4 Software Switch Performance

In this section we are going to analyze the software switch performance. The used switch [29] can be configured in both stateless and OPP stateful configuration. To make a comparison we decided to evaluate the impact of duplicating traffic and the impact of introducing the stateful part. The evaluation of performance was done using iPerf [40] a testing tool useful to test TCP throughput. Since our application deals only with TCP traffic this tool is perfect for the testing we want to execute. We performed a session of 25 repeated tests, with 120 parallel connection started together (using iPerf *-p* option that allows to generate parallel connection). On Host 1 we started the iPerf server and on Host 2 the iPerf client that connects to the server (Figure 4.1). We decided to open a large quantity of parallel connection because our stateful switch takes statistics on a flow basis. For this particular test, we suppose that the DPI can be directly connected on a port of the switch as in the topology in Figure 4.1.

We tested everything in the Mininet Ubuntu VM. To limit the influence of the external machine environment everything was tested in a dedicated machine.

Figure 4.15: Performance of the switch

In Figure 4.15 we can see the results obtained. On the x-axis we can find the 4 switch configurations and forwarding policy we considered:

- **SL 0**: stateless switch, no forwarding to the DPI is performed;
- **SL ALL**: stateless switch, it replicates all the traffic to the DPI;
- **SF 0**: stateful switch, no forwarding to the DPI is performed;
- **SF 20**: stateful switch, in this case we configured a double direction threshold of 10 packets in both direction (STC and CTS): at most 20 packets per flows were forwarded to the DPI.

We did not test the configuration **SL 20** because the switch cannot count the number of packets per flow with the stateless configuration without the help of the controller.

On the graph we can find the plot of the average of our 25 repeated tests

and the 99% Confidence Interval. We obtained that, as expected, the best performance is obtained with the **SL 0** configuration. This configuration is our reference point on the performance of this software switch. In this case we reached 1139.6 Mbps.

The impact of traffic duplication can be seen in the gap between **SL 0** and **SL ALL**, the impact is around 210 Mbps. Next we evaluated the impact of introducing the stateful table. In this case we made a comparison between **SL 0** and **SF 0**. We saw that the impact of the stateful configuration is very similar to the reduction in performance due to the duplication, we reached a reduction of around 240 Mbps. Another important observation we can make regards the difference between **SF 0** and **SF 20**, the impact of duplicating at most 20 packets per flow is negligible with respect to make no duplication.

From the previous observation we can say that duplicating all traffic has the same impact of introducing smartness into the switch, furthermore, duplicating 20 packets does not further impact on the performance, as can be seen by the overlapped Confidence Interval between **SL ALL** and **SF 20**. If we wanted to implement DPI without smartness on the switch we would need to duplicate all the traffic, instead, with our solution, we can make filtering and statistics collection with no impact on the network element (with respect to duplicating all the traffic) with the advantage of saving network resources (we need to send less traffic to the DPI) and saving computation power into the DPI (it has to analyze less traffic).

4.5 DPI performance

Now we are going to analyze the performance of DPI when inspecting filtered traffic. We used a slightly modified version of the program utilized in the classification accuracy evaluation (Section 4.2) and we evaluated the time needed by

the program to complete the classification (with the *Unix* command "*time*"), taking into account only the *user time* (the time executed by the program in *user mode*). In this case we used a different trace: we used a synthetic trace of about 6.6GB for 20 users and 40 minutes of traffic.

To remove any other bottleneck effect accountable to slower hardware components, we performed the tests on a dedicated machine, by pre-loading the trace in RAM. We programmed the DPI to read the trace 100 times to limit the DPI engine initialization overhead, we then repeated the experiment 25 times and we calculated average and 99% Confidence Interval. To isolate the DPI processing costs, and computing the overhead of doing RAM reads, we also built a program that, using *libcap*, reads 100 times the same trace we fed to the DPI. The evaluation is then performed calculating the difference between the average time values obtained using the complete DPI and the average time needed for the program that only reads the trace. Furthermore, we used the same programming language (precisely C) using the same compiler (gcc 6.2.0) for both the programs. To the best of our knowledge we didn't find other sources of external influence that can interfere with our results.

libpcap	Time [s]	99% CI
Complete trace	47.796	0.415
20 pkts per flows	1.0852	0.029
10 pkts per flows	0.6592	0.017

Table 4.1: Time needed by libpcap to only read the PCAP

In Table 4.1 we can find the time needed by the *libpcap* program to just read the PCAP files.

nDPI	Total DPI execution time [s]	**Inspection time only (difference) [s]**	99% CI (on difference)	% variation wrt Complete trace
Complete trace	227.1564	179.3604	0.581	
20 pkts per flows	21.856	20.7708	0.199	**-88.42%**
10 pkts per flows	18.1124	17.4532	0.059	**-90.27%**

Table 4.2: DPI performance with filtered traces

In Table 4.2 we can find the values obtained from the evaluation previously described.

We can see how the time is reduced when the DPI has to analyze the filtered trace. We reached a reduction of 88.42% of time when we filtered the traffic maintaining only 20 total packets (10 per direction). Taking only 10 total packets (5 per direction) we reached a reduction of 90%. In conclusion we reached a huge reduction of time analyzing the filtered trace, this result is in line with the traffic reduction seen in Section 4.3.

4.6 Advantages and Disadvantages of the Proposed Solution

The proposed solution allows to make efficient filtering and statistics collection directly in the data plane. We have seen with the evaluation performed, we can reach high classification accuracy reducing the amount of traffic that the DPI has to analyze, this leads to a reduction of the computational requirements of the DPI. We also saw that this offloaded task doesn't impact on the network element allowing to make statistics collection and filtering on the data plane without paying anything in network performance.
Moreover the proposed solution, being completely programmable, allows the DPI to be placed wherever we want. We can connect it directly to the switch or place it with the controller.

We can also extend this application specifying other state machines for others type of traffic (for example for UDP traffic as seen in Section 3.2.2) with no additional complexity. We can also make application-aware tasks, exploiting the classification from DPI and the programmability of the data plane achieved coupling directly DPIs output with SDN.

Another advantage of this solution is derived from the large quantity of flows parameters that we can compute directly in the data plane; in this solution we proposed only to measure packets and bytes transferred and start and end timestamp of the flow session, but we can compute as well packets interarrival time and other statistics that can be exploited eventually by the ISPs to make many type of analysis.

With the developed application we can also decouple the forwarding task and the statistics collection, we can activate and deactivate these two functionalities. We may want to make only statistics collection without DPI analysis or only filtering if we are not interested in the flow statistics. Moreover, with this solution we can tune filtering parameters on the two flow directions independently allowing to develop different filtering techniques based on the specific DPI needs.

The drawbacks of this solution is the fact that it is not immediately deployable: we need to have an OPP compatible data plane that nowadays is not present in the market, in the next section, however, we are going to present a Classic OpenFlow compatible solution immediately deployable in OpenFlow based SDN network. Furthermore, statistics gathering by the controller adds a little overhead on the controller itself, and cannot be exploited for real time analysis at the controller without adding a polling procedure that gathers the statistics of flows not yet finished.

4.7 Classic OpenFlow Solution

In this section we are going to propose a *reactive* implementation similar to our proposed solution, but based on Classic OpenFlow. A similar proposition can be found in [15]

Since OpenFlow does not assume stateful data plane, the solution must be

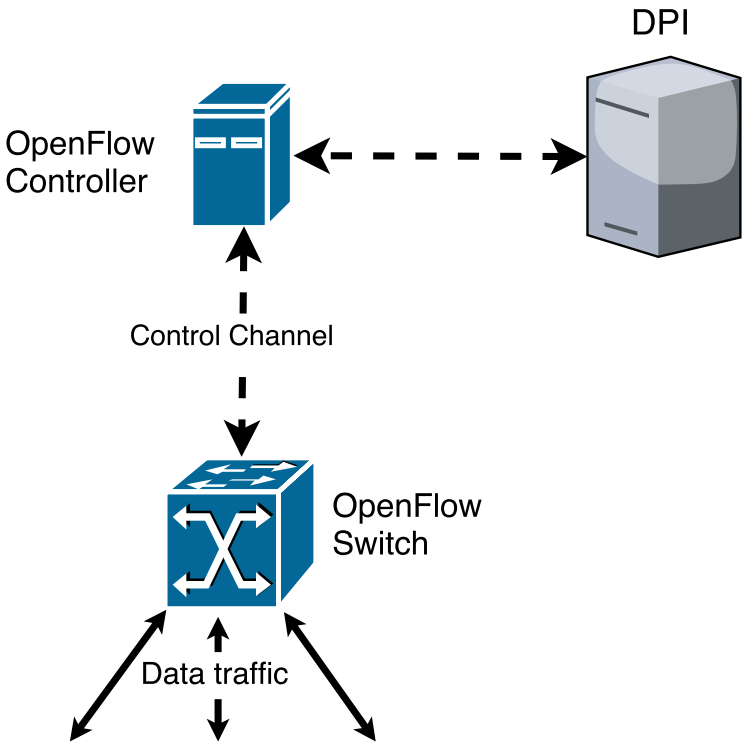

Figure 4.16: Classic OpenFlow Solution

reactive and relying on the direct intervention of the controller. The idea is the following: the controller installs a rule to forward all the traffic of new flows to it, the controller itself will count the number of packets of each flow and when the threshold is reached it will install a rule to block the forwarding of packets from the switch. The controller will forward the packets received from the switch to the DPI that will implement the classification. This scheme can be optimized: the DPI can signal the controller when it has completed the classification, then the controller can install the rule to block the forwarding from the switch even before the pre-determined threshold.

4.7.1 Pros and Cons

The proposed *reactive* solution can be easily deployed with already installed OpenFlow switches. It doesn't introduce modification into the data plane, it requires only to integrate the DPI into the controller or to create a ded-

icated channel between DPI and controller, and to develop the program on the controller. This can reduce the computational requirements of the DPIs. Statistics collection can be, in part, also done into the data plane. OpenFlow allows counting number of packets and quantity of bytes that matches a precise rule, unfortunately it does not allow to calculate other values or make other arithmetic operations on flow parameters.

This approach poses also severe scalability issues caused by the exchange of packets between the switch, the controller, and the DPI. The controller has to manage all the new flows, the first packets of flows, count and redirect them to the DPI, this task can be unfeasible with large quantity of traffic. To make an example, in the CAIDA trace previously analyzed we measured 15.32K new flows per second on average. This value will require an accurate sizing of the controller architecture.

Chapter 5

Conclusions and Future Improvements

This work demonstrated the possibility of exploiting SDN stateful data plane to delegate filtering and statistics collection to the switches. We implemented a prototype on top of OPP, exploiting the possibility of defining an Extended Finite State Machine directly on the fast-path to deal with TCP traffic. We also presented some extensions (i.e. UDP State Machine) and the possibility of exploiting this solution to make application-aware forwarding coupling directly SDN and traffic classification. Finally, we also presented few use-cases in which this solution can fit best.

This work provided the following contributions:

- We showed that traffic filtering can lead to zero-classification accuracy loss, by choosing a proper filtering threshold, confirming the results in [5, 12, 13].
- We evaluated the effect of per-flow sampling on the quantity of data that needs to be analyzed by the DPI, both in terms of bytes and packets, discovering that, even with large thresholds granting 100% classification accuracy, significant gains can be obtained (up to 99% traffic reduction).
- We analyzed the offloading impact on the network elements, showing

that the delegated functionalities allow the software switch to reach high performance comparable to the one obtained duplicating all the traffic.

- We analyzed the computational impact also on the DPI itself, observing that it works more efficiently when traffic is filtered.

- Lastly, we also gave insights on the feasibility to implement the proposed solution in OPP hardware prototype, making it possible to scale the analysis to wire-speed.

To further enhance the proposed solution we would like to explore the following research directions:

- We plan to conduct further experiments with larger traces, possibly collected in a core telco network, and comprising traffic generated by a larger number of users.

- The solution was tested on a virtualized environment. We project to test this solution on a real machine with an accelerated version of the software switch, or otherwise to implement an hardware prototype. This can allow to better analyze the scalability potentiality of the solution.

- The proposed solution collects the number of packets, quantity of bytes, start and end timestamp of each flow on the data plane. We may explore the possibility to compute other flow statistics, adding OPP support for more complex arithmetic operations directly computed on the network data plane.

Bibliography

[1] Sandvine Incorporated, *A global internet phenomena spotlight.* [Online]. Available: `https://www.sandvine.com/resources/global-internet-phenomena/internet-traffic-encryption.html`.

[2] Google, *Google transparency report.* [Online]. Available: `https://www.google.com/transparencyreport/https`.

[3] Z. A. Bahajji, *Indexing HTTPS pages by default*, 2015. [Online]. Available: `https://security.googleblog.com/2015/12/indexing-https-pages-by-default.html`.

[4] L. Baruh, E. Secinti, and Z. Cemalcilar, "Online privacy concerns and privacy management: A meta-analytical review", *Journal of Communication*, vol. 67, no. 1, pp. 26–53, 2017.

[5] N. Cascarano, L. Ciminiera, and F. Risso, "Improving cost and accuracy of DPI traffic classifiers", in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ACM, 2010, pp. 641–646.

[6] M. Mueller, "DPI technology from the standpoint of internet governance studies: An introduction", *School of Information Studies, Syracuse University, Technical Report*, 2011.

[7] C. Xu, S. Chen, J. Su, S. Yiu, and L. C. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms", *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2991–3029, 2016.

[8] G. Finnie, "The role of DPI in an SDN world", *White paper, Heavy Reading*, 2012.

[9] *Audible Magic Website*. [Online]. Available: `https://www.audiblemagic.com/`.

[10] Sandvine Incorporated, *Application zero-rating: Considerations and best practices for fraud prevention*. [Online]. Available: `https://www.sandvine.com/resources/whitepapers/application-zero-rating-considerations-and-best-practices-for-fraud-prevention.html`.

[11] M. Finsterbusch, C. Richter, E. Rocha, J.-A. Muller, and K. Hanssgen, "A survey of payload-based traffic classification approaches", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 1135–1156, 2014.

[12] S. Fernandes, R. Antonello, T. Lacerda, A. Santos, D. Sadok, and T. Westholm, "Slimming down deep packet inspection systems", in *INFOCOM Workshops 2009, IEEE*, IEEE, 2009, pp. 1–6.

[13] J. Khalife, A. Hajjar, and J. E. Dıaz-Verdejo, "Performance of OpenDPI in identifying sampled network traffic.", *JNW*, vol. 8, no. 1, pp. 71–81, 2013.

[14] *nDPI official website*. [Online]. Available: `http://www.ntop.org/products/deep-packet-inspection/ndpi/`.

[15] T. Zhang, A. Bianco, P. Giaccone, S. Kelky, N. Mejia Campos, and S. Traverso, "On-the-fly traffic classification and control with a stateful SDN approach",

[16] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful openflow applications inside the switch", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[17] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification", in *Proceedings of the 2006 ACM CoNEXT conference*, ACM, 2006, p. 6.

[18] G. Gómez Sena and P. Belzarena, "Early traffic classification using support vector machines", in *Proceedings of the 5th International Latin American Networking Conference*, ACM, 2009, pp. 60–66.

[19] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[21] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open packet processor: A programmable architecture for wire speed platform-independent stateful in-network processing", *arXiv preprint arXiv:1605.01977*, 2016.

[22] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model", in *Proceedings of the 30th international Design Automation Conference*, ACM, 1993, pp. 86–91.

[23] *OPP prototype repository.* [Online]. Available: `https://github.com/beba-eu/`.

[24] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "SDN-based application-aware networking on the example of youtube video streaming", in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, IEEE, 2013, pp. 87–92.

[25] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-awareness in SDN", *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 487–488, 2013.

[26] Y. Li and J. Li, "Multiclassifier: A combination of DPI and ML for application-layer classification in SDN", in *Systems and Informatics (ICSAI), 2014 2nd International Conference on*, IEEE, 2014, pp. 682–686.

[27] C. Cascone, R. Bifulco, S. Pontarelli, and A. Capone, "Relaxing state-access constraints in stateful programmable data planes", *arXiv preprint arXiv:1703.05442*, 2017.

[28] *BEBA project home page.* [Online]. Available: `http://www.beba-project.eu/`.

[29] *BEBA software switch implementation.* [Online]. Available: `https://github.com/beba-eu/beba-switch`.

[30] *CPqD OpenFlow 1.3 software switch project home page.* [Online]. Available: `http://cpqd.github.io/ofsoftswitch13/`.

[31] *BEBA controller implementation.* [Online]. Available: `https://github.com/beba-eu/beba-ctrl`.

[32] *Ryu controller website.* [Online]. Available: `https://osrg.github.io/ryu/`.

[33] *Mininet network emulator home page.* [Online]. Available: `http://mininet.org/`.

[34] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks", in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ACM, 2010, p. 19.

[35] *Controller code repository.* [Online]. Available: `https://git.io/vSI9H`.

[36] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: An experimental investigation of QUIC", in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ACM, 2015, pp. 609–614.

[37] *Dpkt python library.* [Online]. Available: `https://github.com/kbandla/dpkt`.

[38] *Official libpcap website.* [Online]. Available: `http://www.tcpdump.org/`.

[39] *The CAIDA UCSD anonymized internet traces - chicago 2015-02-19.* [Online]. Available: `http://www.caida.org/data/passive/passive_2015_dataset.xml`.

[40] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP.* [Online]. Available: `https://iperf.fr/`.