Flow monitoring in Software-Defined Networks: Finding the accuracy/performance tradeoffs

José Suárez-Varela[a], Pere Barlet-Ros[a,b]

[a] *UPC BarcelonaTech (Spain) - {jsuarezv,pbarlet}@ac.upc.edu*
[b] *Talaia Networks (Spain)*

Abstract

In OpenFlow-based Software-Defined Networks, obtaining flow-level measurements, similar to those provided by NetFlow/IPFIX, is challenging as it requires to install an entry per flow in the flow tables. This approach does not scale well as the number of entries in the flow tables is limited and small. Moreover, labeling the flows with the application that generates the traffic would greatly enrich these reports, as it would provide very valuable information for network performance and security among others. In this paper, we present a scalable flow monitoring solution fully compatible with current off-the-shelf OpenFlow switches. Measurements are maintained in the switches and are asynchronously sent to a SDN controller. Additionally, flows are classified using a combination of DPI and Machine Learning (ML) techniques with special focus on the identification of web and encrypted traffic. For the sake of scalability, we designed two different traffic sampling methods depending on the OpenFlow features available in the switches. We implemented our monitoring solution within OpenDaylight and evaluated it in a testbed with Open vSwitch, using also a number of DPI and ML tools to find the best tradeoff between accuracy and performance. Our experimental results using real-world traffic show that the measurement and classification systems are accurate and the cost to deploy them is significantly reduced.

1. Introduction

Traffic monitoring is a cornerstone for network management and security, as it provides an essential information for some tasks such as security policy enforcement, traffic engineering or troubleshooting. With the advent of the Software-Defined Networking (SDN) paradigm, it is even more important to perform a fine-grained monitoring to optimally exploit the possibilities offered by a centralized control plane, which can make decisions with a global view of the network.

Nowadays, one of the most deployed solutions in legacy networks for network monitoring is NetFlow/IPFIX. There are plenty of tools in the market based on NetFlow that harness the information present in flow-level measurements to infer some useful information about the network. These tools provide a wide variety of services such as anomaly detection, Distributed Denial-Of-Service (DDoS) detection or bandwidth monitoring.

Regarding the SDN paradigm, OpenFlow [1] has become a dominant protocol for the southbound interface (between control and data planes). It permits to maintain in the switches records with flow statistics and includes an interface that enables to retrieve this information passively or actively, which could be used to provide similar reports as those of NetFlow.

An inherent issue in SDN is its scalability. Thus, for a proper design of a monitoring system, it is of paramount importance to consider the network and processing overheads to store and collect the flow statistics. On the one hand, since controllers typically manage a large amount of switches in the network, it is important to reduce the controllers' load as much as possible. On the other hand, the most straightforward way of implementing per-flow monitoring with OpenFlow is by maintaining an entry for each flow in a table of the switch. Thus, monitoring all the flows in the network results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources (i.e., the number of TCAM entries and processing power) [2].

Additionally, flows in the measurement reports are often labeled (e.g., by protocol) using port-based classification techniques. However, these techniques are becoming increasingly obsolete, since they are not well suited to current scenarios. Nowadays, it is becoming more frequent to find very diverse applications sharing the same port (e.g., web-based applications) or using non well-known ports to avoid being detected (e.g., P2P applications). Particularly, we can find behind the HTTP and HTTPS ports a wide variety of applications ranging from some services that are very sensitive to delays (e.g., VoIP) to other services whose performance relies on the average bandwidth (e.g., cloud storage). This reflects the necessity of a more comprehensive level of classification where the system can provide information about the specific applications generating the traffic (e.g., Netflix).

Regarding the latest trends in the research area of traffic classification, two main lines can be mainly remarked. On the one hand, techniques based on Deep Packet Inspection (DPI) analyze the packets' payload to identify the traffic. Typically, they can perform an exhaustive classification (e.g., at the application layer) achieving high accuracy levels. Nevertheless, performing DPI over every packet in the traffic is resource consuming and not feasible in all scenarios. On the other hand, other solutions based on machine learning (ML) were proposed to alleviate the burden of the classification based on DPI. These techniques are able to achieve similar accuracy to DPI tools when classifying the traffic by application-level protocols (e.g., SMTP, DNS). However, they cannot accurately identify the applications (e.g., Gmail, YouTube) generating traffic over the same application protocol (e.g., HTTP). For these cases, DPI typically far outperforms ML classifiers.

In the light of the above, we present a scalable monitoring solution with OpenFlow that implements flow sampling and performs flow-level traffic classification with special emphasis on the identification of web and encrypted traffic. As in NetFlow/IPFIX, for each flow sampled, we maintain a flow entry in the switch which records the duration and the packet and byte counts. Moreover, our system efficiently combines some state-of-the-art traffic classification techniques used in legacy networks to provide labeled flow records with the application that generated the traffic. In more detail, our monitoring system has the following novel features:

Scalable: We address the scalabity issue in two different dimensions: (i) to alleviate the overhead for the controller and (ii) to reduce the number of entries required in the flow tables of the switches. To these end, we designed two sampling methods which depend on the OpenFlow features available in current off-the-shelf switches. We implement flow sampling because it is easier to provide without requiring modifications to the OpenFlow specification, although we also plan to provide a packet sampling implementation in a future work. We remark that our methods only require to initially install some rules in the switch which will operate autonomously to discriminate randomly the traffic to be sampled. To the best of our knowledge, there are no solutions in line with this approach. For example, iSTAMP [2] proposes to sample traffic based on an algorithm that ”stamps” the most informative flows. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic report of the flows in the network.

Fully compliant with OpenFlow: Our monitoring system implements flow sampling using only native features present since OpenFlow 1.1.0. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. Furthermore, for backwards compatibility, we also propose a less effective monitoring scheme that is compatible with OpenFlow 1.0.0. Unlike NetFlow in legacy networks, OpenFlow enables to independently monitor specific slices of the network, which can be highly interesting in emerging SDN/NFV scenarios. We found in the literature some monitoring proposals for SDN that rely on different protocols than OpenFlow. For instance, OpenSample [3] performs traffic sampling using sFlow, which is more commonly present than NetFlow in current SDN switches. However, we consider sFlow to have a high resource consumption as it sends every sampled packet to an external collector and maintains there the statistics. In contrast, our system maintains the statistics directly in the switches and retrieves them when the flow expires. The flow entries used by our system are independent from other entries installed by other modules performing different network functions (e.g., forwarding). This allows our system to operate transparently and specifically select the most adequate timeouts to obtain accurate measurements.

Traffic classification: Our system performs flow-level classification combining some state-of-the-art techniques. In particular, we apply specific DPI or ML techniques to different types of traffic according to their trade-off between accuracy and cost. Similarly to [4] and [5], we use information in HTTP headers and certificates of encrypted (SSL/TLS) connections to unveil the applications hidden behind web and encrypted traffic. Moreover, we process the DNS traffic as a complementary source of information to discover the domain names associated to the different flows. This, in turn, can help to identify the applications. For the different classification techniques we evaluate, we individually measured the accuracy against a ground truth and the cost to deploy them in OpenFlow-based SDN environments.

The remainder of this paper is structured as follows: Section 2 shows the architecture of our flow monitoring system. Section 3 describes our flow measurement system. Section 4 describes our traffic classification system. In Section 5, we evaluate the accuracy and overhead contribution of our monitoring system in a testbed with Open vSwitch, with an implementation within OpenDaylight [6] and combining different traffic classification techniques. Section 6 summarizes the related work. Lastly, in Section 7 we conclude and mention some aspects for future works.

2. Architecture of our monitoring system

In Fig. 1 we show the architecture of our flow monitoring system, which is implemented in the SDN controller (i.e., the control plane). We divided it in two different logical subsystems called "Flow measurement system" and "Flow classification system". The former is in charge of installing flow entries in the OpenFlow switches to perform traffic sampling and maintain per-flow statistics. In this way, when a flow entry in a switch expires, this system asynchronously collects the measurements (packets and bytes counts, and duration) of the corresponding flow. Note that we identify the flows by its 5-tuple. At the same time, the flow classification system manages to identify the application which generated each flow in the traffic. To this end, this system combines different Deep Packet Inspection (DPI) and machine learning techniques for traffic classification and eventually provides reports with labels associated to each flow. Thus, our monitoring system combines the outputs of these two subsystems to finally provide a report that includes, for each flow in the traffic, its associated measurements and a label identifying the application which generated it.

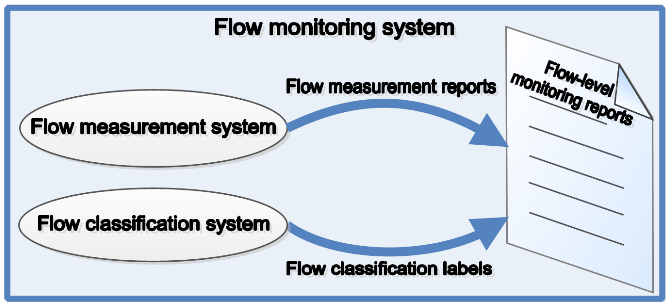

Flow monitoring system
Flow measurement reports
Flow-level monitoring reports
Flow measurement system
Flow classification system
Flow classification labels

Figure 1: Architecture of our flow monitoring system.

The following two sections of this paper provide a detailed description of the flow measurement system (Section 3) and the flow classification system (Section 4) that compose our monitoring system.

3. Flow measurement system

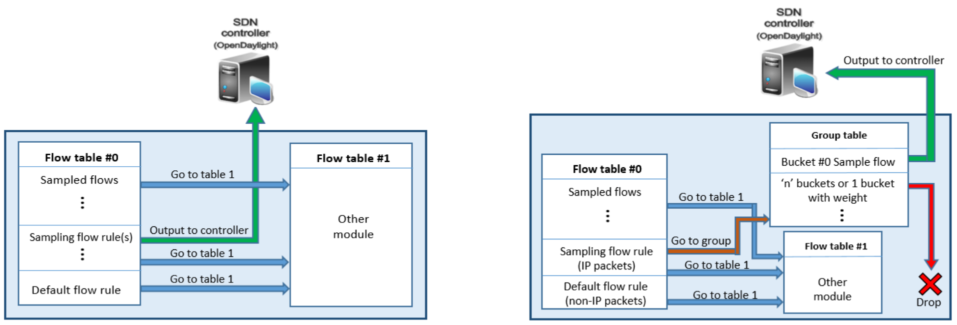
Our flow measurement system fully relies on the OpenFlow specification to obtain flow-level measurements similar to those of NetFlow/IPFIX in legacy networks. This is not new in SDN, since some works, such as [7], used a similar approach earlier. However, to the best of our knowledge, no previous works proposed OpenFlow-based methods to implement traffic sampling and provide reports in a NetFlow/IPFIX style, i.e., randomly sampling the traffic and maintaining per-flow statistics in separated records, which are finally reported to a collector. Since we are aware that OpenFlow has many features that are classified as "*optional*" in the specification, we designed two different sampling methods with different levels of requirements of features available in the switch. These methods, in summary, consist of installing a set of entries in the switch which allow us to discriminate the traffic to be sampled. Thus, we only send the first packets of those flows to be monitored and the controller is in charge of installing reactively specific flow entries to maintain the flow measurements.

Our measurement system makes use of the multiple tables feature of OpenFlow, which is available from OpenFlow 1.1.0. Nevertheless, we propose an alternative solution with some limitations for switches with OpenFlow 1.0.0 support (more details in Section 3.2). The support of multiple tables allows us to decouple the sets of entries of different modules operating in the SDN controller that perform other network tasks.

Before showing the details of our sampling methods, we describe the generic structure of OpenFlow tables in our system, which is illustrated in Fig. 2a. In both methods proposed, the monitoring system operates in the first table of the switch, where the pipeline process starts. In this way, our system installs in this table some entries to sample the traffic and maintains records for monitored flows. All the entries in the first table have at least one instruction to direct the packets to another table, where other modules can install entries with different purposes (e.g., forwarding). Focusing on the table where our system operates, three different blocks of entries can be differentiated by their priority field. There is a first block of flow-level (5-tuple) entries that act as flow records. Then, a block of entries with lower priority defines the packets to be sampled. And lastly, we add a default entry with the lowest priority which directs to the next table the packets that did not match any previous entries. The key point of this system resides on the second block of entries, where the methods described below install rules to define which packets are sampled. The operation mode when a new packet arrives to the switch is to check firstly if it is already in one of the per-flow monitoring entries. If it matches any of these entries, the packets and bytes counters are updated and the packet is directed to the next table. If not, it goes through the block of entries that define whether it has to be sampled or not. If it matches one of these, the packet is forwarded to the next table and to the controller (Packet In message) to add a specific entry in the first block to sample subsequent packets of this flow. Finally, if the packet does not match any of the previous rules, it is simply directed to the next table.

3.1. Proposed sampling methods

We present here the two methods devised for our measurement system and discuss the OpenFlow features required for each of them. One is based on hash functions, which performs flow sampling very accurately, and the other one, based on IP suffixes, is proposed as a fallback mechanism when it is not possible to implement the previous one. Our sampling mechanisms are covered by the Packet Sampling (PSAMP) Protocol Specification [8],



(a) Sampling based on IP suffixes (b) Sampling based on hash function

Figure 2: Scheme of OpenFlow tables and entries of our flow measurement system.

which is compatible with the IPFIX protocol specification. According to the PSAMP terminology, the first method matches the definition of hash-based filtering. While the second method can be classified as property match filtering, where a packet is selected if specific fields within the packet headers are equal to a predefined set of values. We assume that the switches have support for OpenFlow 1.1.0 and later versions so, they have at least support for multiple tables. However, in Section 3.2, we make some comments about how to implement an alternative solution with OpenFlow 1.0.0.

3.1.1. Sampling based on IP suffixes

This method is based on performing traffic sampling based on IP address matches. To achieve it, the controller adds proactively one entry with match fields for particular IP address ranges. Typically, in traditional routing the matching of IP addresses is based on IP prefixes. In contrast, we consider to apply a mask which checks the last n bits of the IPs, i.e., we sample flows with specific IP suffixes. In this way, we sample a more representative set of flows, since we monitor flows from different subnets (IP prefixes) in the network. In order to implement this, it is only necessary a wildcarded entry that filters the IP suffixes desired for source or destination addresses, or combinations of them. To control the number of flows to be sampled, we make a rough consideration that, in average, flows are homogeneously distributed along the whole IP range (we later analyze this assumption with real traffic in Section 5.1.1). As a consequence, for each bit fixed in the mask, the number of flows sampled will be divided by two with respect to the total number of flows arriving to the switch. We are aware that typically there are some IPs that generate much more traffic than others, but this method somehow permits to control the number of flows to be monitored. Furthermore, if we consider pairs of IPs for the selection, instead of individual IPs, we can control better this effect. In this case, if we sample an IP address of a host which generates a large number of flows, only those flows which match both source and destination IP suffixes are sampled. Generically, our sampling rate can be defined by the following expression:

$$sampling\, rate = \frac{1}{2^m \cdot 2^n} \tag{1}$$

Where 'm' is the number of bits checked for the source IP suffix and 'n' the number of bits checked for the destination IP suffix.

This method is similar to host-based (or host-pair-based) sampling, as we are using IP addresses to select the packets to be sampled. However, host-based schemes typically provide statistics of aggregated traffic for individual or group of hosts. In contrast, we sample the traffic by single or pairs of IP suffixes, but provide individual statistics at a flow granularity level. Moreover, to avoid bias in the selection, the IP suffixes can be periodically changed by simply replacing the sampling rule(s) in the OpenFlow table.

To implement this method, the only optional requirement of OpenFlow is the support of arbitrary masks for IP to check suffixes, since there are some switches which only support prefix masks for IP. We also present and evaluate in a technical report [9], an alternative method based on matching on port numbers for switches that do not support IP masks with suffixes, but this method requires a larger number of entries to sample the traffic.

3.1.2. Hash-based flow sampling

This method consists of computing a hash function on the traditional 5-tuple fields of the packet header and selecting it if the hash value falls in a particular range. To implement this method, we make use of the group table feature of OpenFlow. In OpenFlow, a group table contains a number of buckets which, in turn, are composed by a set of actions. Therefore, if a bucket is selected, all its actions will be applied to the packet. For the implementation of this method, we leverage the use of the *select* mechanism to balance the load between different buckets within a group. The bucket selection depends on a selection algorithm (external to the OpenFlow specification) implemented in the switch which should perform equal or weighted load sharing among buckets. In Fig. 2b, we can see the tables structure designed for this method. In this case, all IP packets are directed to the next table as well as to a group table where only one bucket sends the packet to the controller to monitor the flow, other buckets drop the packet. To control the sampling rate, we can select a weight for each bucket. This method much better controls the sampling rate, as we can assume that a hash function is homogeneous along all its range for all the flows in the switch. In contrast to the previous one, this method accurately follows the definition of flow sampling, i.e., sample the packets of a set of flows with some probability.

This method requires the use of group tables with *select* buckets and to have an accurate algorithm in the switch to balance the load among buckets.

3.2. Modularization of the system

Our measurement system leverages the support of multiple tables to isolate its operation from other modules performing other network functions. Thus, we can see our monitoring system as an independent module in the controller which does not interfere with other modules operating in other tables. In the controller we can filter and process the Packet In messages triggered by entries of our module, since these messages contain the table Id of the entry which forwarded the packet to the controller. Note that, in order to process packets in the monitoring system before other modules, it is necessary to properly select the ordering of the Packet In listeners of the different modules operating in the controller. Additionally, our system can be integrated in a network using a hypervisor (e.g., CoVisor [10]) to run network modules in a distributed manner in different controllers.

Alternatively, we propose a solution for those switches that have only support for OpenFlow 1.0.0, where only one table can be used. Since this version does not support group tables, only the method based on matches of IP suffixes can be implemented. Thus, it is feasible to install the monitoring entries by combining them with the correspondent actions of other modules at the expense of loosing the decoupling of our monitoring system.

3.3. Statistics retrieval

To collect flow measurements with OpenFlow, two different approaches can be highlighted. On the one hand, pull-based mechanisms consist of making active measurements, i.e., sending queries (OFPT MULTIPART REQUEST message) to the switch. The switch will respond with an OFPT MULTIPART REPLY message with the requested flow statistics (duration, packets count and bytes count). On the other hand, push-based mechanisms consist of collecting measurements asynchronously. In this case, when adding a new flow entry, idle and hard timeouts are defined. Then, when a flow entry expires, the switch sends to the controller an OFPT FLOW REMOVED message with the flow statistics. Our system envisions a push-based approach to retrieve statistics. Given that it uses specific flow entries, we can selectively choose the timeouts. Thus, we overcome the issue of other push-based solutions like FlowSense [11], where flows with large timeouts are collected after too long a time decreasing the accuracy of the measurements.

4. Flow classification system

Our flow classification system provides labels that identify the applications generating the different flows that were sampled by the flow measurement system (Sec. 3). To this end, it operates in the SDN controller (i.e., the control plane) to combine a number of classification techniques already present in the literature for legacy networks. In particular, we adapt these techniques to implement them in SDN-based networks and perform traffic classification with two different levels of detail: (i) we use Deep Packet Inspection (DPI) and machine learning (ML) techniques to classify flows at the level of application layer protocols (e.g., RTP, DHCP, SSH), and (ii) apply specific DPI techniques aimed at discovering the names of the applications (e.g., Netflix, Facebook, YouTube) generating web and encrypted traffic.

In OpenFlow-based networks, when installing flows reactively, packets belonging to the same flow are sent to the controller until a specific entry is installed for them in the switch. As a consequence, the SDN controller can receive more than one packet for each flow to be monitored. In particular, this occurs during the time interval when the first packet of a flow arrives to the switch, and the time when a flow entry for this flow is installed in the switch. This time interval is mainly the result of the following factors: (i) the time needed by the switch to process an incoming packet of a *new* sampled flow and forward it to the controller, (ii) *Round-Trip Time* (RTT) between the switch and the controller, (iii) the time in the controller to process the Packet In and send an order to the switch to install a new flow entry, and (iv) the time in the switch to install the new flow entry. The first and fourth factors depend on the processing power of the switch. The RTT depends on some aspects like the distance between the switch and the controller or the capacity and utilization of the control link that connects them. The second factor depends on the processing power and the workload of the controller and, of course, its availability. As a consequence, the controller will receive the first few packets of each flow sampled by our flow measurement system. Thus, our classification system leverages this issue by applying DPI only to those packets, which often contain enough information to properly classify the traffic. This allows us to perform accurate classification without producing additional large overheads.

In Fig. 3, we show the operation mode of our classification system when a new packet arrives to the SDN controller. Firstly, it discerns whether the

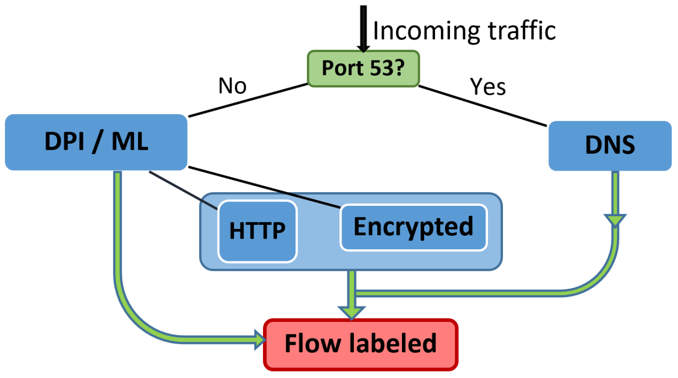


Figure 3: Scheme of our flow classification system.

packet contains DNS traffic or not. We consider that the traffic is DNS if the packet header matches port 53. In this case, the packet is forwarded to our "DNS module" to extract some data from the records in DNS queries. This enables to then apply the method in [5], which, in summary, consists of associating domain names to the server IPs of HTTP and encrypted flows. Otherwise, if the packet is considered as non-DNS traffic, it is processed by our "DPI/ML module", which implements either a DPI tool or a supervised ML model. The aim of this module is to provide a label that classifies the flow at the level of application protocol (e.g., SMTP). Following the scheme in Fig. 3, if the "DPI/ML module" labels the flow as HTTP or encrypted traffic, the packet is accordingly directed to our "HTTP" or "Encrypted" modules. In these modules, we apply DPI techniques proposed in [5] and [4] to extract the hostnames associated to flows either from the HTTP headers or the SSL/TLS certificates. If the "HTTP" or "Encrypted" modules manage to obtain the hostname, the system selects it as classification label for the flow. Otherwise, we use the information collected in the "DNS module" to infer a domain name associated to the server IP address of the flow.

We provide below a more detailed description of the the four classification modules (in Fig. 3) that compose our system:

DPI/ML module: The aim of this module is to classify the monitored flows with labels that identify their associated application-level protocols (e.g., BGP, SNMP). To this end, we implemented within this module two alternative solutions to classify the traffic: one based on DPI, and another based on supervised ML models. We then evaluate these two classifiers (in Section 5.2.2) using real-world traffic to show the tradeoffs between accuracy and resource consumption when applying each of them. Note that, except for those flows that are classified as DNS, HTTP or encrypted traffic, the output label of our classification system is the one generated by this module.

DNS module: In this module, we implemented the proposal in [5], where they use information in DNS queries to then associate domain names to the HTTP(S) flows in the traffic. For this purpose, they rely on the basic assumption that, prior to execute an HTTP(S) request, the client application typically resolves the IP address of the server by sending a DNS query. Hence, monitoring the DNS traffic enables to discover the domain names associated to the server IP addresses of the HTTP(S) flows. Note that, as they highlight in [4], the domain name information is very valuable as it often permits to unveil the name of the application that generated a specific flow. In order to implement this technique, this module should receive the DNS traffic traversing the monitored switches and maintain the information extracted from DNS records until their expiration time. To this end, we proactively add a flow entry in the switches that redirects to the controller all the traffic matching port 53. We then evaluate (in Section 5.2.2) the cost of deploying this technique in a SDN controller and show that the amount of traffic processed by this module is quite reduced in our evaluation scenario using real-world traffic.

HTTP module: In this module, we implement the technique they propose in [5] to extract the hostname from the *host* field in the HTTP headers. In this way, inspecting the first few packets of an HTTP connection is typically sufficient to find the hostname associated to the HTTP flow. This, in turn, provides useful information to discover the applications generating different flows of HTTP traffic. Note that this module only processes the packets that are classified by the "DPI/ML module" as HTTP traffic.

Encrypted module: In this module we implement the technique proposed in [4], where they perform DPI to extract the *Server Name Indication* (SNI) fields of the SSL/TLS certificates exchanged during the handshake prior to establish an encrypted connection. Similarly to the previous module, this information is typically present in the first few packets of these flows and is useful to unveil the applications generating encrypted flows. Note that this module only processes the packets classified as encrypted traffic by the "DPI/ML module".

We provide in Section 5.2 a description of the tools and features involved in the implementation of each of these modules within the SDN controller.

In a nutshell, our approach is to combine different classification techniques and apply them only to specific types of traffic regarding the tradeoff between accuracy and performance. This allows us to achieve high levels of accuracy considerably saving the processing power needed in the SDN controller to perform such a comprehensive classification. Remark that our

Trace dataset	# of flows	# of packets	Description
MAWI [12] 15th July 2016	3,299,166 (total flows) 2,653,150 (TCP flows) 646,016 (UDP flows)	54,270,059	1 Gbps transit link of WIDE network to the upstream ISP. Trace from the samplepoint-F. Average traffic rate: 507 Mbps
CAIDA[13] 18th February 2016	2,353,413 (total flows) 1,992,983 (TCP flows) 360,430 (UDP flows)	51,368,574	This trace corresponds to a 10 Gbps backbone link of a Tier1 ISP (direction A - from Seattle to Chicago). Average traffic rate: 2.9 Gbps
UNIVERSITY-1 25th November 2016	2,972,880 (total flows) 2,349,677 (TCP flows) 623,203 (UDP flows)	75,585,871	10 Gbps access link of a large Spanish university, which connects about 25 faculties and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS). Average traffic rate: 2.41 Gbps
UNIVERSITY-2 17th March 2017	4,679,374 (total flows) 3,712,431 (TCP flows) 966,943 (UDP flows)	298,860,479	This trace was captured from the same vantage point that the trace labeled as "UNIVERSITY-1". Average traffic rate: 3.17 Gbps

Table 1: Summary of the real-world traffic traces used in our experiments.

system provides a deep insight of the traffic, as it not only classifies the traffic by application protocols, but also provides useful information to discover the applications that generate web and encrypted traffic.

5. Experimental evaluation

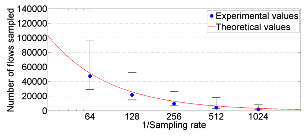
This section includes an evaluation of the two subsystems that compose our Flow Monitoring System. We first evaluate the Flow Measurement System in Section 5.1. Then, in Section 5.2, we evaluate the Flow Classification System regarding its classification accuracy and its deployment cost.

5.1. Evaluation of the Flow Measurement System

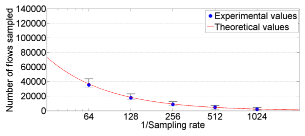
For the evaluation of our monitoring system, we implemented it in a module within the OpenDaylight SDN controller [6]. We conducted experiments in a small testbed with an Open vSwitch, a host (VM) which injects traffic into the switch and another host which acts as a sink for all the traffic forwarded. All the experiments make use of real-world traffic from three different network scenarios. We provide in Table 1 a description of the traces we used. These traces were filtered to keep only the TCP and UDP traffic.

5.1.1. Accuracy of the sampling methods

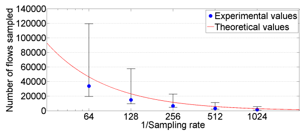
We conducted experiments to evaluate the accuracy of our flow measurement system. Thus, we first assess if the sampling rate is applied properly and if the selection of flows is random enough when using the proposed sampling methods. All our experiments were separately done for the MAWI, CAIDA and UNIVERSITY-1 traces described in Table 1 and repeated applying sampling rates of 1/64, 1/128, 1/256, 1/512 and 1/1024. For the method based on IP suffixes, we considered two different modalities: matching only a source IP suffix, or matching both source and destination IP



(a) MAWI trace

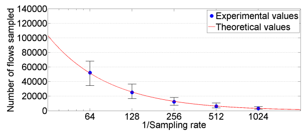


(b) CAIDA trace

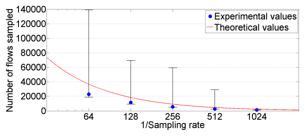


(c) UNIVERSITY-1 trace

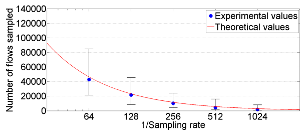
Figure 4: Evaluation of sampling rate for methods based on source IP suffixes.



(a) MAWI trace

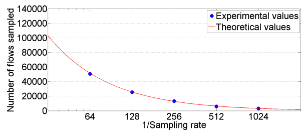


(b) CAIDA trace

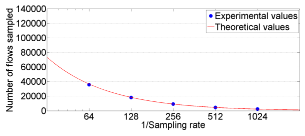


(c) UNIVERSITY-1 trace

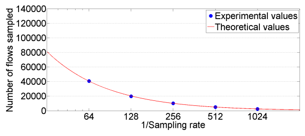
Figure 5: Evaluation of sampling rate for methods based on pairs of IP suffixes.



(a) MAWI trace



(b) CAIDA trace



(c) UNIVERSITY-1 trace

Figure 6: Evaluation of sampling rate for the hash-based method.

suffixes. For each of these modalities, with a particular trace, and a specific sampling rate, we performed 500 experiments selecting randomly IP suffixes.

To analyze the accuracy in the application of the sampling rate, we evaluate the number of flows sampled by our methods and compare it with the theoretical number of flows if we used a perfectly random selection function. We calculated these theoretical numbers by simply multiplying the total number of flows in the traces by the sampling rate applied. We show in Fig. 4, the results for the method based only on source IP suffixes for the three traces. These plots display the median value of the number of flows sampled for the experiments conducted in relation to the sampling rate applied. The experimental values include bars which show the interval between the 5^{th} and the 95^{th} percentiles of the total 500 measurements obtained for each case. Likewise, in Fig. 5, we show the same results for the case that considers pairs of source and destination IP suffixes. Given these results, we can see that the median values obtained are quite close to the theoretical values, i.e., in the average case these methods apply properly the sampling rate established. However, we can see there is a high variability among experiments. This means that, depending on the IP suffixes selected,

Modality of the sampling method		**Number of bits checked**				
		SR = 1/64	**SR = 1/128**	**SR = 1/256**	**SR = 1/512**	**SR = 1/1024**
Based on src IP suffixes	bits src IP suffix (m)	6	7	8	9	10
Based on pairs of src/dst IP suffixes	bits src IP suffix (m)	3	4	4	5	5
	bits dst IP suffix (n)	3	3	4	4	5

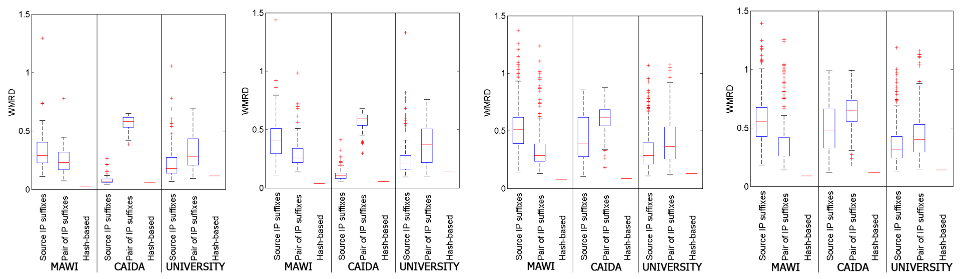
Table 2: Number of bits checked for the source and destination IPs in our experiments.

we can over- or under-sample. Regarding Equation 1, we show in Table 2 the number of bits we checked for the source and destination IP suffixes to define the desired sampling rates in the different experiments we made.

Next, we evaluate the hash-based sampling method making use of the load balancing algorithm for group tables included in Open vSwitch. In line with the scheme in Fig. 2b, we installed in the switch a group table with two buckets of type *select* (with actions *output to controller* and *drop* respectively) and properly defined their weights to send to the controller the desired amount of flows according to the sampling rate applied. The results in Fig. 6, show that this method considerably outperforms the previous one in terms of control of the sampling rate. Not only it samples a number of flows very close to the ideal one, but also it does not experience any variability among experiments as it is based on a deterministic selection function. Furthermore, it achieves good results for the three traces, which indicates that it is a robust and generalizable method for any network scenario.

In order to evaluate the randomness in the selection of our sampling methods, we compare our results with those obtained with a perfect implementation of flow sampling, with a completely random selection process. Thus, if our implementation is close to a perfect flow sampling implementation, the flow size distribution (FSD) should remain unchanged after applying the sampling, i.e., the distribution of the flow sizes (in number of packets) must be very similar for the original and the sampled data sets. We acknowledge that this property is not completely preserved for the IP-based method, but we follow this approach to measure how random is the flow selection of this method and compare it with the hash-based method.

We quantify the randomness of the sampling method by calculating the difference between the FSDs of the original and the sampled traffic. For this purpose, we use the *Weighted Mean Relative Difference* (WMRD) metric proposed in [14]. Thus, a small WMRD means that the flow selection is quite random. In Fig. 7, we present boxplots with the results of our proposed methods. For the sake of brevity, we do not show the results applying a sampling rate of 1/256, which are very similar to those displayed (these results are available in a technical report [9]). We can observe that these results are in line with the above results about the accuracy controlling the

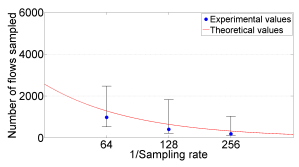


(a) Sampling = 1/64 (b) Sampling = 1/128 (c) Sampling = 1/512 (d) Sampling =1/1024

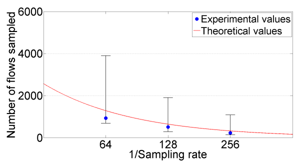
Figure 7: Weighted Mean Relative Difference (WMRD) between FSDs.

sampling rate. The method which shows better results is the hash-based one. Moreover, for the methods based on IP suffixes, we see that for the MAWI trace, the method based on pairs of IP suffixes achieves a more random flow subset. While for the CAIDA and UNIVERSITY-1 traces, the method based on source IP suffixes behaves better. Note that we chose the FSD to compare the randomness of these two methods because the FSD is known to be robust against flow sampling.

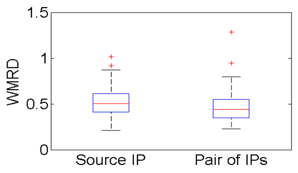
All the previous experiments were done using traffic traces captured in edge nodes of large networks, which aggregate great amounts of traffic from a large number of different subnets within the networks. However, we find also interesting to evaluate how our sampling methods behave when they are deployed in inner switches in the network that forward traffic from few subnets (i.e., IP prefixes). To this end, we repeated the experiments with a filtered trace obtained from the UNIVERSITY-2 trace (in Table 1) that keeps only traffic belonging to a specific department in the university network. In particular, this trace contains traffic from three different IP prefixes with mask lengths of 24 bits. The resulting trace contains a total of 82,183 different flows. For the sake of brevity, we only show some results for the methods based on IP suffixes. However, we also made the evaluation applying the hash-based method and, similarly to the results in Figs. 6 and 7, it achieves a number of flows sampled very close to the theoretical ones with quite low WMRD values. Since the total number of flows in the trace is quite low, we made only experiments applying sampling rates of 1/64, 1/128 and 1/256, as lower sampling rates result in an excessively small amount of flows sampled. Figs. 8a and 8b show the results regarding the accuracy applying the sampling rate respectively for the methods based on source IP suffixes and pairs of src/dst IP suffixes. We also evaluated the results regarding the randomness of these two methods. Fig. 8c shows the results when applying a sampling rate of 1/128. We also made this evaluation



(a) Source IP suffixes



(b) Pair of IP suffixes



(c) WMRD with SR=1/128

Figure 8: Results with traffic from a specific department in trace UNIVERSITY-1.

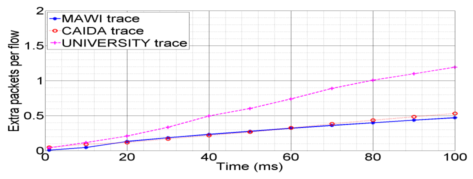
applying sampling rates of 1/64 and 1/256, and obtained similar results.

As final remark, we would like to highlight that our system permits to consistently sample the traffic along the whole network by installing rules in all the monitored switches using the same IP suffixes or, for the hash-based method, sampling traffic that matches the same range of the hash value. This, for instance, enables to perform Trajectory Sampling [15], which is very useful to observe the trajectories of different flows traversing the network.

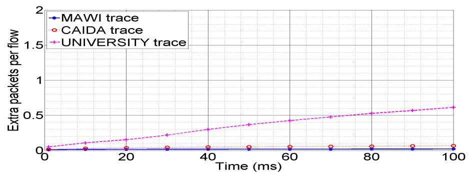
5.1.2. Overhead of the Flow Measurement System

An inherent problem in OpenFlow is that, when we install flows reactively, packets belonging to the same flow are sent to the controller until a specific entry for them is installed in the switch. This is a common problem to any system that works at flow-level granularities. As a consequence, in our system we can receive in the controller more than one packet for each flow sampled. Specifically this occurs during the time interval between the reception of the first packet of a flow in the switch, and the time when a specific entry for this flow is installed in the switch. The factors that contribute to the duration of this time interval were already discussed in Section 4.

In order to analyze all the different bottlenecks in a single metric, we measure the number of packets that are sent to the controller for each flow sampled before the switch installs specific rules to maintain the measurements. That is, the amount of additional packets of the same flow that our measurement system has to process. In the remainder of this section we refer to these additional packets as “extra packets”. We consider a scenario with a range from 1 ms to 100 ms for the elapsed time to install a new flow entry. As a reference, in [16] they observe a median value of 34.1 ms for the time interval to add a new flow entry with the ONOS controller in an emulated network with 206 software switches and 416 links. Thus, we simulate this range of time values for the MAWI, CAIDA and UNIVERSITY-1 traces (in Table 1) and analyze the timestamps of the packets to calculate, for each flow, how many packets are within this interval and, thereby, would be sent to the controller. We analyze separately the overhead for TCP and UDP, as

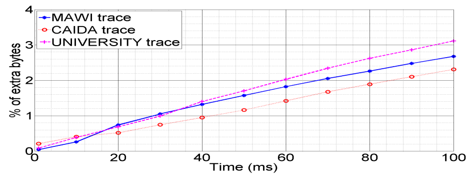


(a) TCP traffic

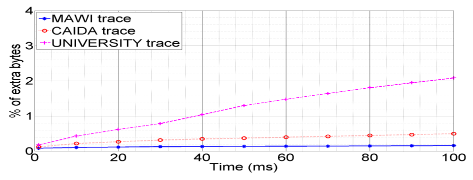


(b) UDP traffic

Figure 9: Average number of extra packets per flow.



(a) TCP traffic



(b) UDP traffic

Figure 10: Percentage of extra bytes.

their results may differ due to their different traffic patterns. We show the results in Fig. 9. As we can see, the average number of extra packets varies from less than 0.2 packets for delays below 20 ms, to around 1.2 packets for an elapsed time of 100 ms with TCP traffic.

Likewise, in Fig. 10 we show the results in terms of average percentage of extra bytes sent to the controller. That way, the percentage of extra bytes ranges from less than 0.8% for elapsed times below 20 ms to 3.1% in the worst case with an elapsed time of 100 ms and TCP traffic. These results show that the amount of extra traffic sent to the controller is significantly smaller than if we implemented the trivial approach of forwarding all the traffic to the controller or a NetFlow probe and not installing in the switch specific entries to process subsequent packets and maintain per-flow statistics.

These results also reflect that, for the UDP traffic, the number of extra packets and bytes per flow is significantly smaller than for TCP flows. Among other reasons, this is due to the fact that typically many UDP flows are single-packet (e.g., DNS requests or responses). Note that these results are not applicable to any scenario, as we can find networks using protocols such as VXLAN, LISP or QUIC that typically generate large flows over the UDP protocol. For these cases, the overhead contribution of the UDP flows would be closer to the case of TCP traffic. That is the case of the results for the UNIVERSITY-1 trace, where we could observe that UDP flows had a larger average number of packets, as is reflected in Figs. 9b and 10b.

From these results, it is possible to infer the CPU cost of running our monitoring system in a SDN controller, as the processing cost per packet can be considered constant. In particular, the controller only needs to maintain a hash table to keep track of those packets sent to the controller and thus not accounted for in switch (i.e., extra packets shown in Fig. 9).

As for the memory overhead in the switch, we implement sampling methods that provide mechanisms to control the number of entries installed. With our solution it is necessary to maintain a flow entry for each individual sampled flow. Thus, there are three main factors which determine the amount of memory necessary in the switch to maintain the statistics: (i) the rate of new incoming flows (traffic matching different 5-tuples) per time unit, (ii) the sampling rate selected, and (iii) the idle and hard timeouts selected for the entries to be maintained. The first factor depends specifically on the nature of the network traffic, i.e., the rate of new flows arriving to the switch (e.g., flows/s). It is a parameter fixed by the network environment where we operate. However, as in NetFlow, the sampling rate and the timeouts (idle and hard) are static configurable parameters and the selection of these parameters affects the memory requirements in the switch. In this way, with (2) we can roughly estimate the average amount of concurrent flow entries maintained in the switch.

$$\begin{gathered} Avg.\,entries = R_{flows/s} \cdot sampling\,rate \cdot E[t_{out}] \\ sampling\,rate \in (0,1] \qquad t_{out} \in [t_{idle}, t_{hard}] \end{gathered} \tag{2}$$

Where "$\mathrm{R_{flows/s}}$" denotes the average rate of new incoming flows per time unit, "sampling rate" is the ratio of flows we expect to monitor, and $\mathrm{E[t_{out}]}$ the average time that a flow entry is maintained in the switch.

In order to configure a specific sampling rate, for the method based on IP suffixes we can set the number of bits to be checked for the IP suffix(es) according to (1). Likewise, for the hash-based method, we can set the proportion of flows to be sampled by configuring the weights of the buckets. Regarding the timeouts, the controller can set the values of the idle and hard timeouts when adding a new flow entry in the switch to record the statistics (in the OFPT FLOW MOD message).

To conclude this section, we propose some different scenarios and estimate the average number of concurrent flow entries to be maintained in the switch. The purpose of this analysis is to have a picture of the approximate memory requirement of our Flow Measurement System. To this end, we rely on (2). In our scenarios we consider the three different real-world traces MAWI, CAIDA and UNIVERSITY-1 described in Table 1. Thus, to calculate "$\mathrm{R_{flows/s}}$" for each trace, we divide their respective total number of flows by their duration. Furthermore, we consider two different sampling

Sampling rate	**Trace dataset**	$\mathbf{R}_{\text{flows/s}}$	**Avg. number of flow entries**						
			E[t]=15 s	**E[t]=60 s**	**E[t]=300 s**	**E[t]=600 s**	**E[t]=900 s**	**E[t]=1,200 s**	**E[t]=1,800 s**
1/128	UNIVERSITY-1	9,916	1,162	4,648	23,241	46,481	69,722	92,963	139,444
	MAWI	3,665	429	1,718	8,590	17,180	25,770	34,359	51,539
	CAIDA	21,672	2,540	10,159	50,794	101,588	152,381	203,175	304,763
1/1024	UNIVERSITY-1	9,916	145	581	2,905	5,810	8,715	11,620	17,430
	MAWI	3,665	54	215	1,074	2,147	3,221	4,295	6,442
	CAIDA	21,672	317	1,270	6,349	12,698	19,048	25,397	38,095

Table 3: Estimation of the average flow entries used in the switch.

rates, 1/128 and 1/1024. For the configuration of the timeouts, we envision a typical scenario using the default values defined in NetFlow: 15 seconds for the idle timeout and 30 minutes (1800 seconds) for the hard timeout. Regarding the average time that a flow remains in the switch ($\mathrm{E}[\mathrm{t}_{\mathrm{out}}]$), we know that it ranges from the idle timeout to the hard timeout. In this way, we consider these two extreme values and some others in the middle. The case with the lowest memory consumption will be when $\mathrm{E}[\mathrm{t}_{\mathrm{out}}]$ is equal to the idle timeout, and the case with the highest consumption, when $\mathrm{E}[\mathrm{t}_{\mathrm{out}}]$ is equal to the hard timeout. The amount of memory for each flow entry strongly depends on the OpenFlow version implemented in the switch. The total amount of memory of a flow entry is the sum of the memory of its match fields, its action fields and its counters. For example, in OpenFlow 1.0 there are only 12 different match fields (269 bits approximately), while in OpenFlow 1.3 there are 40 different match fields (1,261 bits).

Table 3 summarizes the results for all the cases described above. As a reference, in [17] they noted that modern OpenFlow switches have support for 64k to 512k flow entries. To these flow entries estimated, we must add the additional amount of memory of the implementation of the sampling methods described in Section 3.1. For both methods, the switch must allocate an additional table to maintain the sampled flows as well as the entries which determine the flows to be sampled. For the method based on IPs, it uses an additional wildcarded flow entry which determines the IP suffix(es) to be sampled. For the hash-based method, it uses an additional entry to redirect the packets to a group table, as well as the group table with its respective buckets. We don't provide an estimation of this memory contribution since we consider it is too dependent on the OpenFlow implementation in the switch. Nevertheless, we assume that this amount of memory is negligible compared to the amount of memory allocated for the entries that record the statistics of the sampled flows.

Note that our system could be attacked by malicious agents sending malformed packets or messages at a high rate in order to cause congestion in the controller, as it would receive all the first few packets from sampled flows. Also the communication in the Southbound interface could be interrupted because the control channels were compromised. All these are inherent problems of OpenFlow-based networks, since the data plane sometimes relies much on the decision making in the controller. For our particular case, we consider as future work the design of a complementary system to detect and mitigate these types of attacks and mechanisms to guarantee a reliable connection between the data and control planes.

5.2. Evaluation of the Flow Classification System

We implemented our classification system combining different classification tools. For the 'DPI/ML module', we implemented the classifier based on DPI using a distribution of "nDPI" (version 1.8-stable). A list of the protocols supported by this tool is available in [18]. The implementation of our ML classifier was done using the well-known C5.0 decision tree [19], whose code is under the Gnu GPL. We made the ML feature selection based on the work in [20], where they use as inputs for the model some flow-level NetFlow features. In particular, we included the source and destination ports, the IP protocol and the size of the first packets of the flow (with a maximum of 6 packets). As possible classes, we use the application protocols included in the list of classification labels provided by nDPI [18]. To this end, we only selected from this list those labels whose prefix is "NDPI_PROTOCOL", as there are others that identify some specific applications names or type of contents in web traffic for instance. In total, our ML model has 169 different classes. For the implementation of the HTTP, Encrypted and DNS modules, we used respectively the HTTP, SSL and DNS scripts of the open-source tool Bro IDS, which permits to extract the hostname from HTTP headers and SSL/TLS certificates, and the domain names from DNS queries.

5.2.1. Ground truth

In order to evaluate our flow classification system, we created a ground truth using real-world traffic. Our dataset includes a collection of 4,679,374 different flows from the UNIVERSITY-2 trace corresponding to a large university network. More details about this trace are described in Table 1.

We built our ground truth using the open-source tools nDPI and Bro IDS, which also performs DPI. We processed the whole UNIVERSITY-2 trace with both tools to obtain an extensive information about the traffic in the trace. Our ground truth consists of a report including all the flows in the trace (identified by their 5-tuple) and associated labels that classify each of them. As for the selection of the labels, we follow the operation scheme of our Flow Classification System. First we select the label provided by nDPI, which classifies the flows by application protocols. In the case that nDPI classifies a flow as *HTTP* or *SSL/TLS* traffic, we substitute this label for the server hostname extracted either from the HTTP headers or the SSL/TLS certificates. That way, our ground truth includes a collection of labels selected by nDPI except for flows with web or encrypted traffic, where we use the *Fully Qualified Domain Name* (FQDN) of the server associated to the connection. Note that this selection of labels allows us to properly evaluate our classification system for all the techniques it includes. On the one hand, we can compare our ML classifier with the results achieved by DPI, as the ML model produces only output labels included in the list of supported protocols of nDPI [18]. On the other hand, we can evaluate the server domain names obtained by the DNS module comparing them with the hostnames achieved by the HTTP and Encrypted modules.

5.2.2. Accuracy and deployment cost of the Flow Classification system

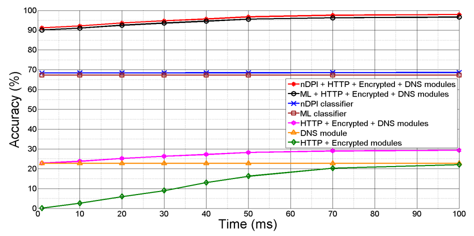
In this Section, we evaluate the different techniques implemented in our classification system in order to better understand their tradeoffs between accuracy and performance when deploying them in SDN environments.

As we discussed in Section 4, in our monitoring system, the controller can receive packets from the same *sampled* flow during the time interval between the reception of the first packet in the switch, and the time when a specific flow entry is installed in the switch. That way, our classification system leverages the reception of all these packets to perform traffic classification.

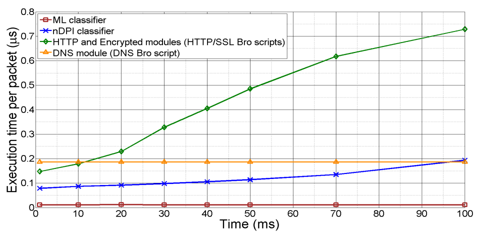
In our evaluation, we considered the same scenario than in Section 5.1.2, with a range from 1 ms to 100 ms for the elapsed time to install a flow entry. Thus, we simulate this range of time values using the UNIVERSITY-2 trace (described in Table 1) and analyze the accuracy achieved and the processing power needed by our classification system. In particular, we analyze these factors separately for each of the modules within the classification system.

In order to evaluate the accuracy achieved by our system, we used the ground truth described in Section 5.2.1. This ground truth represents the results that could be achieved if all the traffic was mirrored to the SDN controller and we applied the DPI techniques in our “DPI/ML” (using nDPI), “HTTP” and “Encrypted” modules to classify every flow. This approach is resource consuming and not feasible in all scenarios. That way, we consider a scenario where we perform DPI only to those first few packets of the flows that the controller receives. Note that, in our scenario, the longer the time elapsed to install a flow entry in the switch, the more packets from the same flow will be received in the controller. Likewise, a larger number of packets processed by our classification system will potentially lead to a higher level of accuracy. Alternatively, we also consider the use of our supervised ML classifier (within the DPI/ML module) that collects flow-level features (described at the beginning of Section 5.2) to perform traffic classification at the level of application protocols. This allows us to compare the results obtained by nDPI with those achieved by ML specifically when classifying the traffic by application protocols. To measure the accuracy of our system, we consider that it classifies a flow correctly if it produces the same label than the ground truth. That is, the application protocol labels produced by nDPI for all the flows except for web and encrypted flows, which are labeled with the server hostname associated. Remark that, for the evaluation of the techniques implemented in our "HTTP", "Encrypted" and "DNS" modules, we only consider the second-level domains of the hostnames. For example, if the label in the ground truth is *www.google.com*, we consider that our system succeeds if it provides the label *google*.

In Fig. 11a, we show the accuracy results (in terms of percentage of flows well classified) individually achieved by our classification modules as well as combining some them. The x-axis represents the elapsed time to install the specific entry for the flow. An important fact we highlight, is that in our "DPI/ML" module we achieve almost the same accuracy either using nDPI or using our ML model. Furthermore, the results achieved for low time intervals show that both techniques are able to classify the traffic even when the controller receives a small number of packets. Note that, it should not be compared the classification performed in this module with a simple port-based classifier. In our ML module, apart from ports and IP protocol, we also include features like the length of the first packets that reach the controller (max. 6 packets). Regarding the operation of nDPI, we do not have specific details of its implementation, but typically DPI tools perform classification not only considering the well-known ports, but also inspecting the payload content. Note that in these particular case, both techniques achieve similar accuracy due to the classification is done at the level of application protocols. Similarly, in [20] we can observe that they achieve very good results using a C4.5 decision tree to classify the traffic in classes with a comparable complexity level. However, we also note that nDPI also provides complementary labels for some flows (e.g., web traffic) that identify the applications (e.g., Facebook) or the type of contents (e.g., AVI contents). In total, it supports 63 different labels of this type. We did not consider these labels in our system as we used more precise DPI techniques (in our HTTP, Encrypted and DNS modules) to specifically unveil the applications generating web and encrypted traffic. Nevertheless, we note that performing such a comprehensive classification by application



(a) Traffic classification accuracy



(b) Per-packet execution times

Figure 11: Evaluation results of our flow classification system.

names is not feasible using ML classifiers without considerably sacrificing the accuracy results. Moreover, in Fig. 11a we also see that the HTTP and Encrypted modules significantly improve their accuracy as the time interval becomes longer. We also remark that, in terms of false positives, the DNS module provides wrong labels for around 21.4% of web and encrypted flows. However, the HTTP and Encrypted modules do not produce any false positive. That is why in our system we prioritize the labels produced by the HTTP and Encrypted modules and use the DNS module as a fallback classifier if the other modules did not achieve an output label.

Regarding the applicability of our classification system for security purposes, we note here some issues that would be addressed as future work. On the one hand, malicious hosts could avoid being monitored if they use for example uncommon ports that nDPI or the ML classifier cannot properly detect. Moreover, the use of encrypted tunnels (e.g., SSH) prevents our system from classifying the different flows that they carry inside.

As for the processing power required to deploy our classification system in a SDN controller, we made an evaluation of the execution cost of the different classification modules. We measure the cost by calculating the average execution time per packet. To this end, we execute some experiments in an ordinary machine with an Intel i7 quad-core processor and 8 GB of RAM memory. We process the whole traffic trace (UNIVERSITY-2) with each module and divide the processing time by the total number of packets in the trace. Note that each module does not process all the packets in the traffic, but only the incoming packets to the module. For example, the DNS module only processes the traffic matching port 53 (more details are described in Section 4). In Fig. 11b, we show the results of the per-packet execution times for the different modules. From these results, we infer that the ML module performs much better than the DPI module in terms of processing overhead, as we expected. Also note that the accuracy and the execution times of our DNS module do not depend on the elapsed time (x-axis) to install the flow entries, since it is always fed by all the DNS traffic independently of this time interval. The reduced per-packet execution times of the DNS module are mainly due to the low amount of DNS packets that can be typically found in real-world traffic. For example, in the trace we used in our experiments, only 0.487% of the total number of packets correspond to DNS traffic.

We remark that the purpose of this evaluation is to compare in relative terms the costs of the different classification modules. Nevertheless, note that the execution times we provide can be inflated if we compare them with those times in real scenarios, as our values include the time to read the traffic trace from the hard-disk memory of the computer.

6. Related work

In this section, we provide an overview of the state-of-the-art regarding traffic measurement and traffic classification in Software-Defined Networks.

There are a number of efforts in the literature to perform traffic measurement in the SDN paradigm. However, this is an issue which still has a lot of room for improvement. For example, in [7] they use the measurement features of OpenFlow to maintain per-flow statistics in the switches. However, their approach is not scalable as they do not perform traffic sampling and it requires to install an entry in the flow tables for every flow in the traffic. In iSTAMP [2], they perform flow-based sampling by making use of a multi-armed-bandit algorithm to “stamp” some flows. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic dataset of the flows in the network. Alternatively, some authors suggest to make use of different architectures specifically designed for monitoring tasks. Thus, in [21], they propose using OpenSketch, where some sketches can be defined and dynamically loaded to perform traffic measurement. We also found some proposals that rely on different protocols than OpenFlow. For instance, OpenSample [3] performs traffic sampling using sFlow. Nevertheless, we consider sFlow has a high resource consumption as it sends every sampled packet to an external collector to maintain there the statistics. Other authors propose distributed solutions to address the scalability issue in SDN. Thus, in OpenNetMon[22], they design an scheme to monitor flows in edge switches and make measurements of throughput, packet loss and delay. However, these solutions may produce a high overhead in the controller, which has to calculate all the flow paths and install as equitable as possible the flow entries in all the edge switches.

Regarding the traffic classification, we could not find in the literature many contributions specifically addressing the SDN paradigm. We highlight for instance the architecture they proposed in [23] to select flow features for traffic classification in OpenFlow-based networks. However, for traditional networks, there are plenty of proposals to classify the traffic using different techniques ranging from Deep Packet Inspection (DPI) to machine Learning (ML). Thus, in our classification system we used some techniques such as the C5.0 decision tree [19] they used in [20] to classify the traffic, or those methods proposed in [4][5], where they use specific DPI techniques addressing web, encrypted and DNS traffic.

7. Conclusions and future work

We presented a flow monitoring system for OpenFlow which provides reports like in NetFlow/IPFIX and it is enriched with labels identifying the application generating each flow. In order to reduce the overhead in the controller and the number of entries required in the switch, we proposed two traffic sampling methods that can be implemented in current OpenFlow switches. For traffic classification, we efficiently combined some DPI and machine learning techniques with special focus on the identification of web and encrypted traffic. We implemented our system in OpenDaylight and evaluated its accuracy and overhead in a testbed with real traffic. As future work, we plan to use the reports of our flow monitoring system to perform automatic network management based on deep learning.

Acknowledgement

This work was supported by the European Union's H2020 SME Instrument Phase 2 project "SDN-Polygraph" (grant agreement n° 726763), the Spanish Ministry of Economy and Competitiveness and EU FEDER under grant TEC2014-59583-C2-2-R (SUNSET project), and by the Catalan Government (ref. 2014SGR-1427).

References

[1] 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, p. 69, 2008.

[2] M. Malboubi, L. Wang, C. N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)Aggregation and Measurement Paradigm (iSTAMP)," *Proceedings of the IEEE INFOCOM*, pp. 934–942, 2014.

[3] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *Proceedings of the International Conference on Distributed Computing Systems*, pp. 228–237, 2014.

[4] M. Trevisan, I. Drago, M. Mellia, and M. M. Munafò, "Towards Web Service Classification using Addresses and DNS," *International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 38–43, 2016.

[5] T. Mori, T. Inoue, A. Shimoda, K. Sato, S. Harada, K. Ishibashi, and S. Goto, "Statistical estimation of the names of HTTPS servers with domain name graphs," *Computer Communications*, vol. 94, pp. 104–113, 2016.

[6] "The OpenDaylight platform," http://www.opendaylight.org/.

[7] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, "Assessing the Quality of Flow Measurements from OpenFlow Devices," *8th International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[8] B. Claise, "Packet sampling (PSAMP) protocol specifications," 2009.

[9] J. Suárez-Varela and P. Barlet-Ros, "Reinventing NetFlow for OpenFlow Software-Defined Networks (Technical report)," *arXiv preprint arXiv:1702.06803*, 2017.

[10] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," *Proceedings of Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015.

[11] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," *Lecture Notes in Computer Science*, vol. 7799 LNCS, pp. 31–41, 2013.

[12] "MAWI Working Group traffic archive - [15/07/2016]," http://mawi.wide.ad.jp/mawi/.

[13] "The CAIDA UCSD Anonymized Internet Traces 2016 - [18/02/2016]," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[14] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[15] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Transactions on Networking*, vol. 9, no. 3, pp. 280–292, 2001.

[16] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, and B. Lantz, "ONOS: towards an open, distributed SDN OS," *Proceedings of HotSDN*, pp. 1–6, 2014.

[17] "Can OpenFlow scale?" https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/, accessed: 14 July 2017.

[18] "nDPI - List of supported protocols," https://github.com/ntop/nDPI/blob/1.8-stable/src/include/ndpi_protocol_ids.h, accessed: 2017-12-05.

[19] "Data Mining Tools See5 and C5.0," https://www.rulequest.com/see5-info.html.

[20] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the impact of sampling on NetFlow traffic classification," *Computer Networks*, vol. 55, no. 5, pp. 1083–1099, 2011.

[21] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *Networked Systems Design and Implementation, (NSDI)*, vol. 13, pp. 29–42, 2013.

[22] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," *IEEE/IFIP NOMS*, 2014.

[23] A. Santos, C. C. Machado, R. V. Bisol, L. Z. Granville, and A. Schaeffer-filho, "Identification and Selection of Flow Features for Accurate Traffic Classification in SDN," *In Network Computing and Applications (NCA)*, 2015.