Space-Efficient Per-Flow Network Traffic Measurement in SDN Environments

André Madeira
andre.madeira@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professor Luís Rodrigues and Professor Nuno Santos)

Abstract. The ability to obtain a characterization of the flows carried by a given networking infrastructure has many applications, such as resource planning, traffic engineering, intrusion detection, denial of service detection, to name a few. However, given the high speed of current networks, this task is extremely challenging, since the amount of traffic that may need to be processed at line speed is very large. In this work we are interested in studying techniques that allow for the extraction of fine-grained characterizations, specifically ones related to packet lengths and inter-arrival times of a few selected flows therefore allowing for their interpretation and the resolution of more complex network problems. For this purpose, we aim at exploring the possibilities open by programmable switches, and by recent advances on space saving data structures, such as data sketches, to achieve a good tradeoff between the accuracy of the information extracted and the efficiency of the data collection procedure.

Table of Contents

1 Introduction

Network traffic measurement is the process of collecting quantitative data regarding the traffic of a network is a fundamental instrument for the operation and security of a network. Simple measurement tasks may include measuring flow sizes and measuring packet loss which may allow for the completion of more complex tasks such as identifying network anomalies, tracking heavy hitters, intrusion detection, denial of service detection, among many others [1, 2].

Traditional network traffic measurement is often supported by technologies like NetFlow [3], which are usually hardware-dependent, tools such as ping and traceroute [4], which are limited in what measurements they can perform, and tools that rely on packet sniffers like WireShark [5] and sometimes even on the SNMP protocol [6]. Due to modern advances in networking, these have been complemented or even replaced by newer tools and technologies, such as software-defined networking (SDN) [7], which allows for complete interoperability within the network, and programmable switches [8], that allow for a greater range of measurements to be performed within the devices themselves. The latter are especially needed because, given the high speed of current networks, the amount of information that may need to be collected, like the average packet size of each individual flow, and processed at line speed, for a given measurement, can be too large for the often-limited amount of memory possessed by switches.

Due to the difficulties in storing and processing large amounts of data at line speed, most network measurements only collect aggregate information, build estimates based on samples, or are limited to collecting just a few traffic features [1]. Older solutions would often implement sampling or use specialized hardware, a middlebox, to aggregate the traffic of multiple network devices to analyze it. Sampling runs the risk of misrepresenting flows it would sample while middleboxes might not be able to efficiently process all incoming traffic from the multiple network devices present in the network, introducing latency and even timing out packets. In this work we are interested in studying techniques to extract per-flow fine-grained characterizations, including packet length distributions and inter-arrival time distributions, among other features. Such fine-grained characterization may be useful for several tasks, namely within the network security domain. Some possible applications of interest are the detection of covert channels embedded in multimedia streams [9] by detecting the perturbations introduced into the packet distribution by said channels and website [10], smartphone app [11] and encrypted video transmission fingerprinting [12].

For this purpose, we aim at combining techniques that have been made viable by recent advances in networking technologies, such as the previously mentioned SDNs and programmable switches, P4 and space saving data structures. SDNs provide the tools to change the behavior of switches at runtime, and therefore to dynamically select the flows for which statistical data is collected. Programmable switches allow to perform simple computations at line speed, at the switches, which opens the door for extracting fine-grained information with almost no added latency. In order to program the switches, we employ the use of the P4 [13] programming language for the target and protocol independence it provides.

Finally, space saving data structures such as bloom filters [14] and counting sketches [15, 16] can be integrated within programmable switches at the packet processing operation to provide ways for the switch to store large amounts of information, in a space efficient manner, as estimates. Space-saving algorithms also allow for the careful management between the space the estimates take up and the loss of accuracy between the estimates and the real values.

In this report, we make an overview of the main techniques that are available to collect a detailed characterization of traffic flows and survey the most relevant works that enable the gathering of measurements with similar levels of granularity to the ones we aim at achieving. Our aim is to investigate the viability of a system that allows for the line-speed, efficient and fine-grained characterization of individual flows that is dependent on the tradeoff between the accuracy of the obtained measurements and the memory space they occupy, when using state-of-the art switches and SDN protocols.

We are interested in performing two main operations: First, a filtering operation that would allow users to dynamically filter traffic to measure by specific protocols and their respective header fields at runtime. This operation leverages SDNs, for users to communicate to the SDN controller what rules to add to the switch, and programmable switches and P4 to target individual fields of both widely-known and custom-made protocols. Second, a measurement operation that allows for the tracking and storage of distributions of packet lengths at the switch that may later be retrieved by the SDN controller. The controller will then additionally calculate a set of summary statistics over packet length and packet inter-arrival time timeseries. Besides SDN, this operation leverages space-saving algorithms, to efficiently store the packet length distributions while preserving their accuracy, and both programmable switches and P4 to implement the space-saving algorithm over the packet processing pipeline.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 discusses several other works in our area. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of collecting, in real time, a fine grain characterization of network flows with minimal signaling, processing, and memory overhead. In particular, we are interested in implementing three components: Programming a space-efficient, low-latency and hardware-independent switch packet processing pipeline, with an integrated space-saving algorithm for facilitating the storage of a potentially large quantity of metrics. A remote software controller that can query the switch for the characterizations and even extract other metrics from them. And an API connected to the controller that enables users, often network analysts of managers, to introduce an extensive amount

of filtering conditions for the passing traffic. As these filtering conditions are introduced, the controller propagates them towards the switch.

> *Goals:* We aim at building a system that can perform on-line analysis of selected traffic flows in order to obtain a characterization of the statistical properties of that flow.

Our work combines features provided by SDN network architectures, programmable switches, P4, and novel space-saving data structures. We build a flexible system that can provide accurate yet space-efficient metrics while reducing the impacting to the switch's regular capabilities as little as possible. At the switch we will collect quantized frequency distributions of packet lengths. These distributions will then be exported to a SDN controller where we will extrapolate a collection of summary statistics over packet length and packet inter-arrival time timeseries, such as maximum, minimum, mean, percentiles, skew and kurtosis. These metrics may be used for application identification, "heavy hitter" detection and traffic change detection. Network analysts and managers can further adapt the system to their specific needs by filtering out any unwanted traffic. The filtering conditions provided to the users, through the API, will center around network protocols and header fields.

In this report we offer a survey of recent advances in each of these areas, with particular emphasis on how they have been used for different network measurement tasks, and identify a set of mechanisms that are relevant for our task. Based on this analysis we propose an architecture to achieve our goals.

The project will produce the following expected results:

> *Expected results:* The work will produce i) a specification of the architecture used to extract flow characteristics; ii) an implementation of the proposed architecture, iii) an extensive experimental evaluation using a fully operational prototype to identify covert Skype traffic, as seen in Barradas et al. [9] study, as a concrete application example.

3 Background

In this section we provide the required background for our work. We start by making a brief introduction to the problem of performance network measurement and to the main techniques that have been used to implement it. Additionally, we provide examples of characterizations that network measurement tasks often target. Then, we perform a brief overview on the recent advances in networking, including to software-defined networking (SDN), programmable switches, and languages to program those switches such as P4. Finally, given that many network measurement tasks may consume large amounts of storage, we also look at recent advances on space-saving data structures.

3.1 Network Traffic Measurement

Network traffic measurement is the process of extracting quantitative information regarding the operation of a network, including data regarding the operation of the devices and links, but also information about the traffic that the network carries. Typically, network measurement is performed with the help of the network devices themselves, that keep some statistics regarding their own operation. The values captured by the network devices can be read remotely, using protocols such as SNMP (Simple Network Management Protocol [6]) or HTTP, such that they can be collected and analyzed in a central location. Application level tools that run on the network endpoints, such as *ping* and *traceroute* [4], that exploit native features of the IP protocol can also be used to extract useful information, such as reachability and packet routes.

Unfortunately, the type, amount, and level of detail of information collected by outdated tools such as NetFlow [3], which has to resort to sampling to manage the passing traffic, can be limited and not enough to satisfy the requirements of all modern network management tasks such as: detecting which flows consume more resources in a network (also known as "heavy hitters" [17]), detecting denial of service attacks [18], intrusion detection [19], among others. For a comprehensive survey of measurements that are relevant in today's network, the reader is referred to the works of Callado et al. [1] and Mohan et al. [2].

One way of obtaining detailed information regarding the flows, a sequence of packets from a source to a destination, that pass through the network consists in collecting the packets and logging them at some central location for deferred processing. Packets can be collected using specialized hardware and middleboxes to collect traffic traces, sniffers like WireShark [5] (that copy packets as they cross the network) or by having switches duplicate the packets and send copies to a central location [20]. Unfortunately, this approach is not without limitations. Current small and medium sized networks can route packets at speeds in the range of billions of packets per second. Logging packets at this speed may be unfeasible and consume too many resources. This can be mitigated using strategies such as sampling [21], where information regarding only a random subset of the packets collected, with some loss of accuracy. Sampling is also difficult to apply when one seeks to collect information regarding short flows, as sampling may collect few, or even no samples, of the target flows.

It is worth to notice that some network measurement task can be executed by collecting information at a single point in the network, while others require information to be collected at different points and then aggregated before it can be processed. Examples of the first class of tasks include packet size and UDP/TCP protocol ratio. Examples of the second class of tasks include packet delay and packet drop rate. In this work we are mainly concerned with measurements that can be performed by capturing information at a single network device. This is the case of flow characterization, that can be achieve by extracting information from the packets that are forwarded by network switches.

There are a large variety of metrics that can be taken by different network traffic measurement tasks:

Network-Level Measurements At the network-level, network performance metrics like throughput, packet loss and packet delay are commonplace for works in this area. This is mostly due to this area of network traffic measurement being more focused on overall network debugging and optimization.

Flow-Level Measurements At the flow-level however, we can see some more diversity in the metrics taken. Some works, such as [22, 23], focused on the UDP/TCP ratios and TCP/IP options, seeking to provide a better understanding of usage patterns regarding these protocols. Murray et al. [22] sought out to investigate which was the dominant transport layer network protocol, TCP or UDP, in terms of flows, packets and bytes and allowed for the better understanding of network usage patterns employed by streaming applications. Zhang et al. [23] study intended to promote further development on the IP and TCP protocols by reflecting on the most common characteristics of Internet traffic as well as point out misbehaviors and errors when employing these protocols.

A study by John et al. [24] aims at achieving accurate Internet statistics by discovering the mean and distribution of packet sizes, as well as a range of network and transport layer statistics. This is similar to our choice of metrics.

If we were to try and collect packet length distributions with only these techniques, the optimal way to do so would be through the addition of a middlebox to the network between the switches/routers and the users. Despite the heavy resource consumption, it is our best alternative to collect detailed metrics, since the use of sampling could lead to a misrepresentation of the traffic.

3.2 Networking Advancements

As we discussed in the previous section, the amount and detail of the information traditional network traffic measurement tasks can collect is limited and often require the use of sampling or packet collection. However, recent advances in networking architectures and hardware offer new avenues for performing fine-grain measurements without resorting to sampling or packet collection.

Software-Defined Networks Software-defined networking (SDN) [7] is an architecture that allows for the underlying infrastructure of networks to be abstracted for applications and network services. It decouples the network control from the forwarding functions of network devices. It enables the network control, what decides if and where packets are forwarded, to become directly programmable, facilitates network management, and enables efficient network configuration in order to improve network performance and monitoring.

The SDN architecture is known for separating what is called the data plane and the control plane. The control plane usually refers to the part of the networking stack that makes decisions about where traffic is sent and the data plane is the part through which user packets are transmitted/forwarded. The control plane is managed by what is known as a SDN controller. An SDN controller manages several devices of the SDN network, interacting with the devices to control

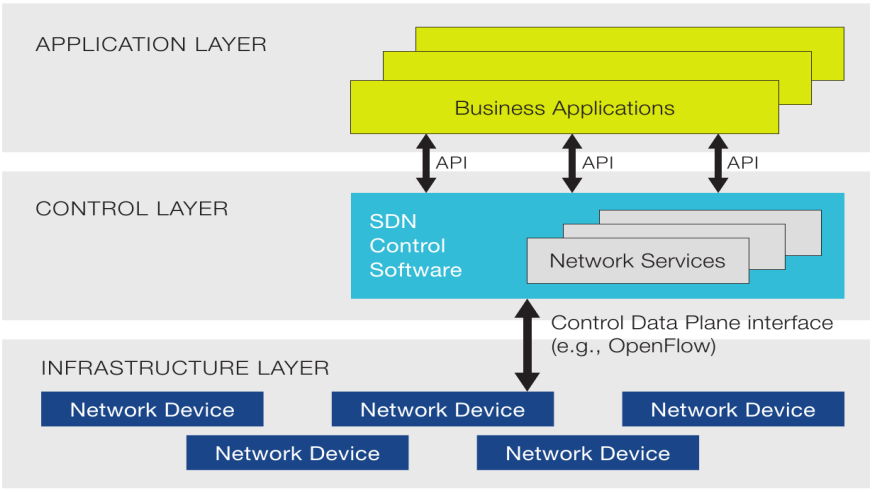

Fig. 1. SDN networking stack. From the SDN white paper [7]

their behavior. Overall, the SDN architecture, often represented as a stack as seen in Figure 1, contains a total of three layers: the application, control, and infrastructure layers. The infrastructure layer consists of the networking devices, such as switches and routers. The control layer holds the SDN controllers, which usually operate on separate machines and servers. The application layer holds the applications, that use the controller to request network resources and to define the network behavior. An SDN controller communicates with the network devices through an API called the *southbound API* (the name derives from the location of the API in the networking stack) and with the services and applications through an API called the *northbound API*. The application can use this interface to modify the configuration and operation of the network. This allows SDNs to be dynamic, manageable, cost-effective, and adaptable.

SDNs introduce several benefits to networking including lower operating costs, easy automation of network devices, improved security across the network, and improved quality of service for multimedia transmissions. We will only focus on two others. First, SDNs provide a centralized view of the entire network, making it easier to centralize measurement gathering from multiple devices. Second, they allow for the repurposing of existing hardware with the use of the SDN controller, also allowing for less expensive hardware to be deployed to greater effect.

SDNs typically operate as follows: First, the controller may pre-configure the switch with a set of permanent rules. Otherwise, when a switch receives a packet that does not match any already existing rule, it must contact the controller in order to handle the new packet. The data plane will clone the packet towards the controller and drop the original. The controller will evaluate the packet and

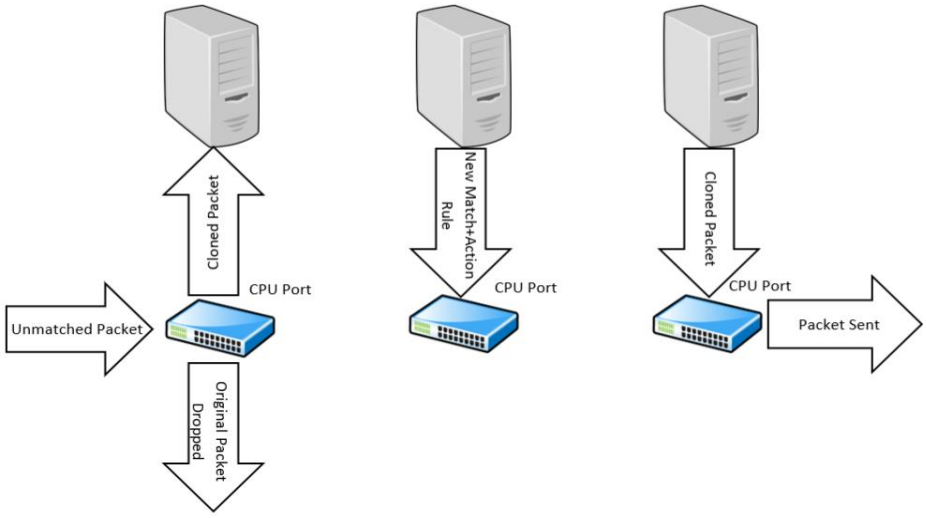

Fig. 2. Process for the addition of new rules.

add a new rule to the switch matching prominent header fields of the packet, including the source and destination MAC address, source and destination IP, source and destination port, protocol, to name a few. Afterwards, the packet will be resent towards the switch where it will be matched to the rule just added and forwarded toward the proper destination. It should be noted that in the case where the controller receives multiple packets, all of the same flow, then only one instance of the rule should be added and all of the resent towards the switch. In this case, rules will usually have a timeout associated with them to flush them after a set period of inactivity. Figure 2 illustrates this process.

Several southbound interfaces have been proposed since the popularization of SDN, among which OpenFlow [25] stands as one of the most relevant.

OpenFlow OpenFlow defines a communication protocol in SDN environments, enabling the SDN Controller to directly interact with the data plane of network devices such as switches and routers. It can query any of the switches it is connected to for a wide range of information it may require, including information about its ports, interfaces, flow structures, statistics and forwarding rules. Additionally, and as we will see in the following section, in the case of programmable switches, for any custom-built structure we may introduce to the switch. It benefits from easy programmability, allowing for the addition of new and innovative features at an accelerated rate, optimized network performance, easy implementation of network-wide forwarding policies and vendor independence.

However, OpenFlow explicitly specifies protocol headers on which it operates. As such, despite originally only supporting four protocols: Ethernet, VLANs, IPv4 and ACLs, nowadays about fifty protocols are supported. And despite grow-

ing quite large already, the community worries if OpenFlow will keep supporting a growing number of protocols increasing the complexity of the specification while still not providing the flexibility to add new headers. Additionally, while it is possible to extend OpenFlow with new features the switches themselves are still limited in what they can accomplish. This is often due to them possessing fixed-function chips designed by the original vendors.

Unlike what we saw in the previous section, if we tried to collect our metrics while making use of SDN technology and OpenFlow, we could immediately make some improvements. First, we could dispose of the middlebox entirely since SDN technology can be easily integrated into preexisting networks. While the middlebox is able to collect the measurements, it might have to handle the traffic of multiple switches by itself. If the middlebox lacks the processing power to handle influx of packets this could quickly lead to packets being delayed and even getting timed out. Instead, by performing the measurements at each individual switch the workload can be split up throughout all the switches. By utilizing switches that support our measurements and extending OpenFlow to do the same, we could simply have the controller use OpenFlow to directly query the switch for the metrics. Additionally, as controllers frequently provide centralized control over the multiple switches, it is possible to perform this on multiple devices with a single controller. However, resource consumption is still a problem, specifically memory usage could quickly escalate to take up most of the available space in the switch. It is also limited when trying to support our traffic filtering operation, since both the switch and OpenFlow might not support network protocols the user requires, whether it be custom-made or otherwise. Not only would the switch be unable to process these protocols, but OpenFlow would be unable to install any new rules concerning them.

Programmable Switches Programmable switches [8] are switches whose behavior can be reconfigured at runtime by programmers.

Regular switches are not reconfigurable by design and, as such, required a fixed protocol like OpenFlow to communicate with the controller. Additionally, in the past, programmable switches could only process packets at a rate of 1/10th or 1/100th compared to fixed-function ones. Today however, there are programmable switches on the market that process packets just as fast as the fastest fixed-function switches. This type of programmable chip is typically called a PISA chip (Protocol Independent Switch Architecture).

With programmable PISA chips it became possible to define how packets are processed in switches. The packet processing capabilities programmable switches enable in the data plane allow for the integration of techniques that allow for the efficient storing of statistics, like sketches for example. With the intent of creating a common programming language for both ASIC and PISA chips, P4 was created. P4 not only provides target independence, but also provides a way for switches to be reconfigured after deployment, all the while not being tied to any number of protocols like OpenFlow.

Once more, programmable switches can aid in performing our measurements. First, due to the packet processing programmability PISA chips provide us, the switch would be able to support any network protocol the user would require. Additionally, we would be able to resolve the high memory usage of the previous solution. By directly altering the packet processing operation we can circumvent this issue. Either by using space efficient storage techniques, which we will discuss soon, to store the metrics until they are requested by the controller or by setting up a flush time loop for the saved metrics with the controller, to periodically dispose outdated metrics. Unfortunately, OpenFlow still fails to support all the necessary network protocols.

P4 The Programming Protocol-Independent Packet Processors language (P4) [13] is a high-level language for programming protocol-independent packet processing and data plane programming. P4 relies on the concept of match+action pipelines. Forwarding network packets can be broken down into a series of table lookups, until a suitable "match" is found, and modifications to protocol headers, which are known as "actions".

P4 has three goals: *reconfigurability*, programmers should be able to change the way switches process packets once they are deployed, *protocol independence*, it should not be tied to any predefined set of network protocols dependent of the switch hardware but any protocol of interest can be easily supported by the data plane programming, and *target independence*, programmers should be able to describe packet processing functionality independently of the specifics of the underlying hardware.

Since P4 addresses only the data plane of a packet forwarding device, it does not specify the control plane nor any exact protocol for communicating state between the control and data planes. Instead, the P4Runtime API, which we will discuss next, can be used. As a whole, P4's goals allow for plenty of optimizations over what programmable switches or OpenFlow (and other protocols similar to OpenFlow) could accomplish for measurement operations on their own.

Since P4 dealt exclusively with the programmable data plane, there was no one way for the control plane to communicate with it. So, p4.org decided to launch the API Working Group, to create silicon-independent APIs for controlling the forwarding plane of switches. P4Runtime's architecture makes it independent of protocols as well as underlying forwarding switch. It lets users control any forwarding plane, regardless of whether it is built from a fixed-function or programmable switch ASIC. Some of its main functionalities are managing match+action tables, which allows for the modification of entries in the match+action tables, and updating the forwarding plane logic. Due to their protocol independence, P4 and P4Runtime can scale to new protocols unlike OpenFlow, as it was designed keeping in mind fixed-function switches. OpenFlow is target independent but protocol dependent. Adding a new protocol involves a lot of time, effort and community involvement. Additionally, while OpenFlow's counters are all tied to the table matching operation, P4 counters can be in-

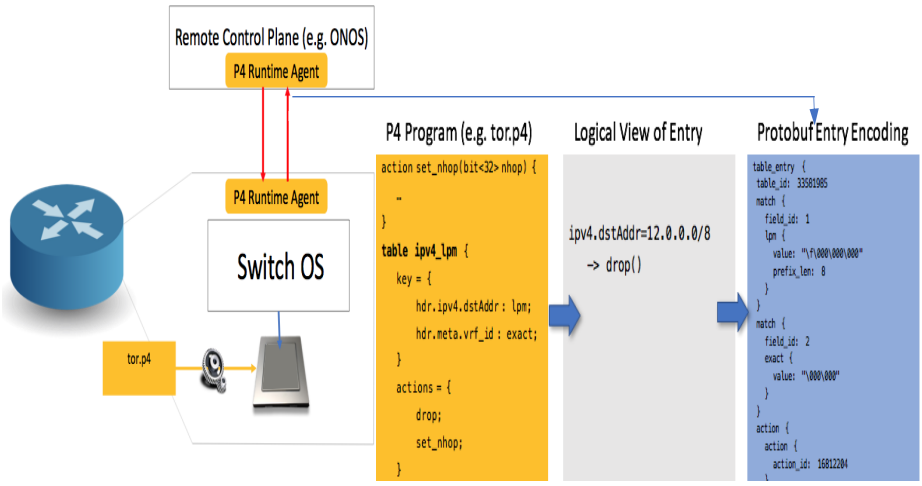

Fig. 3. Communication between a controller running a gRPC client and the data plane running the gRPC server. A table and a logical representation of an entry for the tor.p4 program. P4Runtime encoding of a table entry for the tor.p4 program.[1]

cremented outside of them and tied to certain conditions, like packet size or protocol header fields.

P4Runtime messages are sent between a gRPC client, in the controller, and a server, in the switch. gRPC is a remote call system procedure that allows for subroutines in the server to be executed directly by the client as if it were performing it locally. Figure 3 first shows an overview of the communication between the gRPC client and server and a representation of a table for a P4 program, a logical representation of a table entry for that very same table and a encoding for a table entry in P4Runtime. Each table and table entry as a set of key parameters for matching and a set of actions of which each entry can choose a single one from. In this case, the table has an entry in which the ipv4 destination address is 12.0.0.0 and a metadata field "vrf_id" of 0. The first field is matched by longest prefix first, in this case 8, and the second must be exactly the same. If a packet matches this rule, it will be dropped per the specified action. P4Runtime uses Protobuf (Protocol buffers) as the serialization method for data. The use of both P4 and P4Runtime together make for a less complex alternative to OpenFlow. The use of gRPC allows for easy extensibility, since if new operations are necessary users can simply write the new procedure on the controller, and provides a low-latency, highly scalable alternative over standard communication protocols, since it transports data as binary allowing for quick deployment of the sent procedure.

[1] Available at: https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html

P4 can also help improve our measurements. P4 and P4Runtime allows our solution to become independent of any specific protocols or hardware. Due to how easy they are to extend, by using P4 and P4Runtime it would be simple to add support for whatever protocol the user may require. They also improve upon the data plane programmability of "PISA" chips by allowing a controller to alter the way packets are handled dynamically. Finally, P4Runtime's gRPC messages provide the control plane with a better performing method for communicating with the data plane over OpenFlow.

3.3 Space-Saving Algorithms

Sampling is often used in order to reduce the amount of data a switch would have to store to manageable levels, while still maintaining adequate levels accuracy to perform the necessary measurements, as most switches' storage space is severely limited, even when compared to the storage of average household computers. It is common that in order to properly monitor a network, it is necessary to either use middleboxes explicitly for that purpose or utilizing techniques that minimize the amount of data a switch has to hold (even if temporarily). Sampling is one of these techniques. However, sampling is not without faults, as it can lead to a misrepresentation of the data.

A possible alternative or complementing technique to sampling would be to use algorithms that allow for the data to be stored in as little space as possible, utilizing estimates instead of exact metrics. However, this type of solution often has to consider a tradeoff between their accuracy and how much space they take up. Next, we describe a few types of these algorithms, from more counter based approaches, to the use of more advanced data structures like bloom filters and counting sketches:

Counter Based Techniques These methods usually use individual counters to count the frequencies of different items and hash tables to store the explicit keys of those items allowing it to report the most frequent elements. It is possible to adapt these techniques towards counting operations and making them track all packet flows passing through in order to identify as many as possible. However, the estimated storage space occupied would still surpass those of the other techniques presented here. Counter based techniques are usually focused only on the number of items they store, disregarding implementation overheads like space usage and as such can take up a high amount of memory. Assuming we would need to keep track of 500000 individual counters of 32-bits each, that would require approximately 16 mb of storage.

Bloom Filters A bloom filter is represented as a array of single-bit entries used to test whether an element is a member of a set while balancing a tradeoff between measurement accuracy and computational and storage complexity. False positive matches are possible, but false negatives are not. Elements can be added

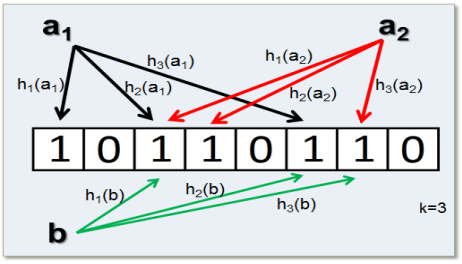

Fig. 4. Bloom filter operation example. [2]

to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

Figure 4 shows the operation of a bloom filter in which three items (a1, a2, b), each represented by three bits, are being inserted into the set by three separate hash functions (h1, h2 and h3). The multiple bits being used are to reduce the number of false positives, as all three bits must be set for an element to be present, which in this case means that b is a false positive. Usually bloom filters are used exclusively to test the membership of a set. However, counting filters provide a way to implement a way to delete elements from the set by turning each array position from a single bit to a n-bit counter. The addition operation now increments the appropriate counters and the lookup operation now checks whether each of the required entries are non-zero. Counting filters can produce similar results to the technique we will present next with the difference that the next technique uses a sublinear number of cells while a counting filters match the number of elements in the set. Usually, counting arrays operate in ranges of less than 5 mb while maintaining accuracy values above 90%.

Counting Sketches Counting sketches keep track of estimates of the frequencies of items that have been observed without the need to maintain keys, identifiers for the items, for each of them, using counters that are incremented with each observation.

The sketch is made up of an array of counters and a set of hash functions that map items into the array. The array is treated as a sequence of rows, and each item is mapped by each of the hashes into their designated row. An item is processed by mapping it to each row in turn via the corresponding hash function and incrementing the counters to which it is mapped. The number of columns in the array is based on a user-defined variable ϵ, that defines the deviation factor between the estimate and the real value. The number of columns is usually much smaller than the number of items that need to be mapped which is why sketches only require a sublinear amount of memory to work. The number of rows is based on a user-defined variable δ, that defines the probability of the error in answering a query be within ϵ. Due to the error introduced with these two variables, when

[2] Available at: https://redislabs.com/blog/rebloom-bloom-filter-datatype-redis/

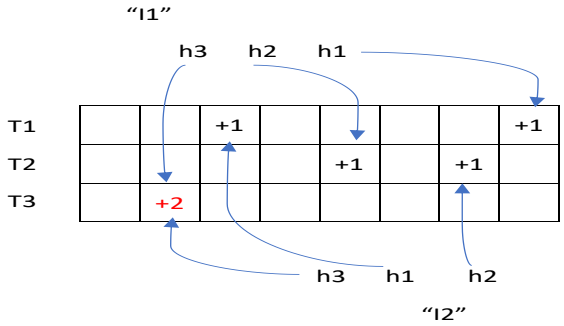

Fig. 5. Counting Sketch operation example.

querying sketches, statistical operations like the median and minimum are used to deduce the closest possible estimate to the true value from the different values of the entries for each item. Ideally, these will produce summaries of the data they hold that are mergeable, meaning that one can process many different portions of traffic independently, and then the summaries computed from each can be quickly combined, for example through union or intersection of them, to obtain an accurate summary of various combinations. This drastically reduces memory usage, compute time, and latency compared to exact methods.

Figure 5 shows the operation of a counting sketch in which two items are being inserted into three different rows of the array by hash functions. All hash functions (h1, h2 and h3) are different from one another and upon selecting the correct entry the counter therein is incremented.

Two examples of counting sketches are the Count Sketch [15] and the Count-Min Sketch [16] (CM Sketch). Both are very similar data structures, each being represented as 2-dimensional array of counters that utilize hash functions to map items to each row. However, CM Sketch's design, which we will discuss in Section 4, makes it a better alternative to Count Sketch as it can achieve similar results in smaller amounts of space.

These methods are typically optimized for hardware implementation as they use statically allocated memory and do not store explicit flow identifiers. Therefore, these methods can answer queries, that when given a flow identifier, generate an estimation for that flow. Usually, counting arrays operate in ranges of several kb while maintaining accuracy values above 90%.

Although all these techniques perform their objective (of saving storage space) they do so to either different degrees or they are optimized for different objectives. More specifically, counters, even with the use of hash tables, would still take up too much space for commodity use, and bloom filters, despite sometimes being able to conserve more storage space than sketches, are optimized for detecting the presence of elements in a set rather than counting and in order

to perform the same tasks as sketches would require the use of multiple bloom filters at a time. As such, counting sketches emerge as a promising alternative for achieving the goal of keeping track of fine-grained traffic metrics.

4 Related Work

In this section, we examine and discuss previous work in the area of network traffic measurement. We start by surveying works that perform network traffic measurement at the network-level, to explore the differences they possess when compared with the type of measurements we are seeking to perform. Next, we look at some works that perform network traffic measurement at the flow-level, to provide an overview of the sort of issues common with the solutions we are focusing on. Afterwards, we dwell on works that leverage applied tasks, such as intrusion detection. We compare the above works across six different properties. Finally, we focus on works related to space-saving algorithms, to compare how different algorithms and data-structures perform and how they could improve the measurement of traffic flows.

4.1 Network-Level Traffic Measurement

In this section we discuss works that perform measurements of the network load across multiple network devices, like packet delay or packet loss between two switches.

OpenSample [26] presents a platform for network measurement in commodity SDNs. It focuses on ensuring low latency by leveraging packet sampling to provide near real-time measurements of both network load and individual flows. OpenSample can be integrated into already existing networks.

sFlow [27] is used to sample packets from an OpenFlow switch. These samples are used to capture two types of data. First, it captures TCP packet header samples with low overhead and extracts TCP sequence numbers from the captured headers to reconstruct nearly-exact flow statistics. Second, packet samples are used to estimate port utilization at sub-second time scales. Finally, a single, centralized collector combines samples from all switches in the network to construct a global view of traffic in the network.

In OpenSample, by sampling the header of TCP packets it is possible to obtain an estimation of the average packet lengths for the passing packets. Additionally, by leveraging the sampling probability and the OpenFlow counters that count how many packets have been through the switch it is also possible to estimate the average inter-arrival time. While it uses SDN technology, which gives OpenSample a centralized view and control over all switches in the network, it is severely limited in other areas. While it would be possible to estimate the packet length distributions using OpenSample, depending on the sampling rate, the results could be inaccurate. This is due to the use of sampling as we run

the risk of misrepresenting the traffic flows. Additionally, we would be limiting ourselves to only being able to measure TCP packets.

We argue that the sampling of entire packet and counters could be avoided in order to avoid misrepresenting traffic flows. Instead the space used to store the wanted metrics could be reduced and queried from the controller directly, especially as several applications may not require sub-second time scales. With this reduction, sampling would no longer be necessary and the additional computation from performing the estimation from the TCP sequence numbers would no longer be required.

OpenNetMon [28] is a network monitoring tool made for use in software-defined networks in OpenFlow. OpenNetMon seeks to provide accurate measurements while reducing network and switch CPU overhead. It does so by increasing the rate at which switches are polled when flows arrive or change frequently and decreases when flows stabilize to minimize the number of queries.

OpenNetMon focuses only on a few select measurements while OpenSample reconstructed near-exact statistics to perform measurements of both the network load and individual flows. However, OpenNetMon's measurements are less focused on counting operations and one of them requires additional packets to be inserted into the network. OpenNetMon is also not limited to TCP traffic.

OpenNetMon is able to provide accurate measurements of throughput, packet loss and packet delay across all flows.

OpenNetMon continuously monitors all flows between predefined link and destination pairs. To determine throughput for each flow, OpenNetMon regularly queries switches to retrieve the amount of bytes sent and the duration of each flow, enabling it to calculate the effective throughput for each flow. Per-flow packet loss is calculated by polling flow statistics from the first and last switch of each path and subtracting the increase of the source switch packet counter with the increase of the packet counter of the destination switch. Path delay is calculated by using OpenFlow's capabilities to inject packets into the network. At every monitored path, packets are regularly injected at the first switch and have the last switch of the path send it back to the controller to estimate the path delay.

Unfortunately, since OpenNetMon is limited by OpenFlow's regular features, it is not possible to perform our specific measurements with it. In order to do so we would have to modify the OpenFlow implementation in use and alter the way the switches function, so that they would both support the counting our packet length distributions. Additionally, the use of sampling could result in the misrepresentation of traffic flows.

While these techniques are effective at calculating some complex statistics like packet transmission delay, since it cannot be measured at any one switch, the storage space occupied by the required information of each flow may prove to be too high. Specifically, every time the controller attempted to query the switch it might have to send several mbs of data across the network. Even if the

switches were queried frequently to avoid this accumulation of data, it might still flood the network. The use of space-saving algorithm would address this issue.

However, OpenNetMon is an improvement when compared to OpenSample, as it performs per-flow, fine-grained measurements and does not capture packets.

Narayana et al. [29] study states that instead of designing piecemeal solutions to work around existing switch mechanisms, the right approach is to co-design language abstractions and switch hardware primitives for network performance measurement. They present a declarative query language that for diverse set of network performance operations that can also be efficiently implemented in switch hardware using a programmable key-value store primitive.

Unlike the previous two works, and most of the other works discussed in these sections, Narayana et al. designed a query language instead of a piecemeal architecture or platform. Their own switch hardware primitives are general enough to perform a variety of measurements while using key-value stores to aggregate information across sets of packets.

Their query language enables network operators to specify diverse performance questions, independent of their implementation on the network. Operators are able to request per-packet performance information, request traffic experiencing uncommon performance, like high queuing delays, aggregate information over packets sharing headers, find simultaneous occurrences of performance conditions and compose queries over results of other queries. Additionally, by implementing a programmable key-value store it is possible to aggregate information across different sets of packets. The value stores and programmatically updates aggregated information across packets belonging to the same key. The key-value stores are kept in cache using a least recently used (LRU) cache-eviction policy.

Additionally, Narayana et al. [29] used programmable switches to access available metadata to better support certain performance-related queries and realize several constructs necessary to perform them.

Seeing as the query constructs are implemented directly within the programmable switch, it would not be too difficult to perform our own measurements with this language, by adding our own if necessary. To obtain inter-arrival time measurements, we would simply count every single packet over a period of time. However, to obtain packet length distributions several key-value stores would be required for each flow. Additionally, the filtering operation is also easily supported by the query language. However, this would lead to either a heavy use of space to maintain several stores for each flow or we would quickly run the risk of losing metrics due to the use of the LRU policy for the key-value store cache. Additionally, since it does not use SDN technology, it may be difficult to perform the measurement over different switches.

While this study provides a general language to perform highly diverse measurement tasks, it does not take into account the heavy storage space usage that could be built up with time, especially when the often small amount of storage space some commodity switches are limited by.

When compared to the two previous works, we can see that it does not require sampling and is not bound to any specific protocol or hardware, however it is not easily integrable as each network device would have to be manually reconfigured every time a change in the network's operation occurred.

4.2 Flow-Level Traffic Measurement

In this section we discuss works that perform measurements of individual flows in network devices, like packet sizes and protocol usage ratios. As a whole, these works have more to do with the traffic passing through the devices and the performance of the devices themselves as opposed to the entire network.

Duffield et al. [30] study discusses several methods on the use of flow statistics formed from sampled packets to infer additional information on the missing unsampled flows. They focus mainly on the accuracy of the estimations of the unsampled flows while minimizing the consumption of resources by the measurement operations. Duffield et al. do not require the capturing of packets.

This work is very similar to OpenSample [26]. They both estimate statistics of unsampled flows through the use of TCP header fields and statistical inference techniques. However, Duffield et al. make even finer use of TCP details and general-flow binomial distributions to perform their estimations, all without capturing packets themselves.

Through the presented methods, they are able to deduce both the number of missing flows that evaded sampling and the distributions of their lengths. This is achieved through statistical inference, and by exploiting protocol-level detail reported in the flow records, more specifically by using the number of flows containing TCP packets with a set SYN flag to estimate the total number of TCP flows. They most commonly obtain flow statistics from Cisco's NetFlow [3] and packet samples from Inmon's sFlow [27], which are then exported from routers to a collector.

It is not possible to perform our measurements using the techniques here as we would run into the same problems we saw in OpenSample. Similarly to OpenSample, using the samples provided would allow us to obtain the packet length distributions. At the same time, we would once again be limiting ourselves to only being able to measure TCP packets, and having to rely on sampling once again. Additionally, it also does not take advantage of SDN technology.

We argue that if the reduction of consumed resources is one of the driving focus of this paper then sampling is unnecessary to accomplish that, data sketches only require sublinear space (a matter of KB at most) and it is feasible that sketches could accomplish the same quality of metrics without additional storage required (perhaps even requiring less storage based on the sampling ratio) all the while avoiding the added complexity incurred from trying to estimate the length of the unsampled flows.

FlowRadar [31] is an attempt at improving NetFlow [3], a feature on Cisco routers that provides the ability to collect and monitor IP network traffic, while removing the need for sampling.

FlowRadar is the only work we discuss which is a direct improvement of an already existing technology, NetFlow. Like OpenSample [26] and Narayama et al. [29], FlowRadar can perform a variety of measurements. However, unlike them, it does not need to perform sampling through the use of a space-saving algorithm to reduce the size of the data obtained from the measurements.

As it is an improvement over NetFlow, FlowRadar is able to provide the same type of records as the original NetFlow including, but not limited to, packet and byte counts for flows, flow duration, layer 3 headers and source and destination ports (if applicable).

FlowRadar can scale up to a large number of flows with small memory and bandwidth overheads by encoding flows and their counters into a small fixed memory size that can be implemented in merchant silicon with constant flow insertion time at switches. The encoded flowset data structure made up of two parts: First a flow filter which is just a normal bloom filter with an array of 0s and 1s, which is used for testing if a packet belongs to a new flow or not. Second is a counting table which is used to store flow counters. This allows the remote collector to perform network-wide decoding and analysis of the flow counters. Thus, all the flows can be captured without sampling and periodically exported to the remote collector in short time scales. Afterwards, given the encoded flows and counters exported from the different switches, the remote collector can perform network-wide decoding of the flows, and temporal and spatial analysis of the flows for numerous different monitoring applications.

Unfortunately, even though FlowRadar can perform a variety of per-flow measurements and even uses a form of space-saving algorithm, it is not possible to obtain our packet length distributions on FlowRadar. This is due to FlowRadar's features not being modifiable to collect our metrics. Additionally, since it does not utilize SDN technology, it would not be able to support our filtering operation since it would not be possible to change switch's behavior at run time.

While FlowRadar provides an excellent platform for the gathering of flows and their metrics without sampling, we argue that its operation could be made even more efficient through the use of sketches to store the required metrics, because, as we will in following sections, sketches require only a sublinear amount of storage. As opposed to their current method which requires both a bloom filter which uses a almost linear number of entries and an additional counting table.

FlowRadar is a big improvement over Duffield et al. [30] study, as besides not requiring packet capturing, it performs per-flow fine-grained measurements, does not use sampling and uses a space-saving algorithm.

UnivMon [32] presents a framework for general-purpose flow monitoring for use in SDN environments which provides high accuracy measurements.

UnivMon is the only we discuss that performs only general-purpose measurements. By collecting some small amount of specific data during the measurement when can, afterwards, extract different statistics from them. Even if the original measurement was performed only for a single statistic. To do reduce the amount of memory utilized, it leverages sketching algorithms. Unlike is also the only work we discuss here that is implemented over P4.

Univmon's data plane uses what are called universal sketches, which have their foundation on the concept of universal streaming. Universal sketches hold multiple sketch instances each performing the same measurement operation, like "heavy hitter" detection. Each subsequent instance may only count a packet if all previous instances have counted it as well, at which point it may sample the packet with some probability. Universal sketches can only perform measurement operations over metrics that satisfy a series of statistical properties. By utilizing universal sketches additional information may be extracted from these measurement operations that could not be extracted otherwise.

The UnivMon control plane queries the data plane and runs simple estimation algorithms for every management application of interest and allows applications to run estimation queries on the collected counters. The control plane generates sketching manifests that specify the monitoring responsibility of each switch. These manifests specify the set of universal sketch instances for different dimensions of interest (IPs, 5-tuples, among others) that each switch needs to maintain for different origin-destination pair paths that it lies on. Periodically, the UnivMon controller gives each switch a sketching manifest which specifies the dimensions for which it needs to maintain a sketch. When a packet arrives at a node, the node uses the manifest to determine the set of sketching actions to apply. When the controller needs to compute a network-wide estimate, sketches, pulled from all nodes and for each dimension, are combined across the network for that dimension. This method minimizes communication to the control plane while still making use of the controller's ability to optimize resource use. An example of a Network Traffic Measurement task that can be accomplished with UnivMon is the detection of change, which is the process of identifying flows that contribute the most to traffic change over two consecutive time intervals. This computation takes place in the control plane, so the output of the universal sketches from multiple intervals can be stored there without impacting online performance. By comparing the volume of flows in two adjacent time frames. If the difference in volume of a flow between the first sketch and the second one is higher than a predefined percentage of total change in all flows, then that flow is said to be a heavy change flow.

Despite UnivMon utilizing space-saving algorithms, SDN technology and being both hardware and protocol independent, it is not possible to perform our measurements over it. This is due to the fact that several fine-grained measurement operations are not supported by universal sketches since they do not meet their requirements. This is especially unfortunate since UnivMon already supports its own filtering operation.

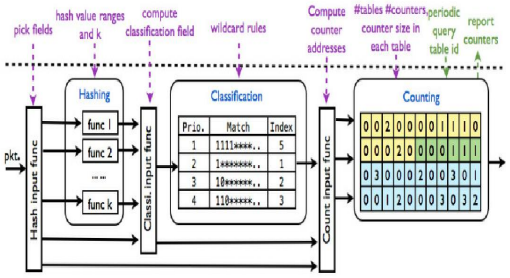

Fig. 6. OpenSketch Data Plane. From the OpenSketch [33] paper.

However, UnivMon is not without faults. Being application agnostic can be useful, however it is doubtful that many applications require focus on all available sub-regions of interest, especially considering that multiple dimensions require multiple sketches which require more memory.

UnivMon not only showcases most of the same properties as FlowRadar but also is easily integrable into network and is not dependent of any specific protocol and hardware. However, unlike FlowRadar, its measurements are not fine-grained and requires multiple instances of the same sketch for some of the measurement tasks.

OpenSketch [33] proposes a traffic measurement architecture with a focus on maintaining a careful balance between measurement generality and efficiency.

OpenSketch can support a wide variety of measurement operations by implementing different space-saving algorithms in the switch and simply altering the amount of memory they require and how they interact with each other. Like some other works discussed here, it utilizes SDN technology, allowing the controller to automatically configure the switch in order to perform the desired measurement operations.

OpenSketch maintains high link speed and low cost by utilizing one of its many sketches optimized towards the required operation or, when some traffic characteristics are not directly supported by sketches, using a simpler sketch and leaving the complex data analysis for the controller.

The data plane has three stages as seen in Figure 6: First, a hashing stage used to reduce the measurement data, which picks the packet source field and calculates a single hash function. Next, a classification stage used to select flows, which picks the packet destination field and filters all the packets matching a rule. Each rule has a index used to calculate the counter location in the final stage. And finally, a counting stage to accumulate traffic statistics, which uses a counting input function to calculate the index of the bit to be updated using the hash value of the source field.

Its data plane architecture allows OpenSketch to be generic and support various measurement tasks but in order to also be efficient and save switch memory

only a few simple hash functions should be used. The OpenSketch controller provides a sketch manager that automatically configures the sketches with the best memory-accuracy tradeoff and a resource allocator that divides switch memory resources across measurement tasks. An example of such a task is the detection of heavy hitters. By employing the Count-Min Sketch to count the volume of flows and a reversible sketch, which would allow it to construct a reversible array to get the keys for each count, to identify the flows in the Count-Min Sketch with heavy counts.

OpenSketch can support our measurements. As it already supports the count-min sketch we can simply design our own measurement task. This task would have to modify three phases of OpenSketch's data plane. First, the hashing stage would need enough compatible hash functions to meet our accuracy requirements. The classification phase in conjunction with the hashing phase would need to separate the packets not only by flow but by size as well. Finally, the counting phase would include have to include enough count-min sketch counters to reduce possible collisions within the sketch. There are two problem with using OpenSketch for our measurements: First, it is not hardware independent, meaning that it adapted to different hardware configurations. Second, it is unable to support our filtering operation to the degree of detail we seek as it is limited to a certain set of network protocols.

OpenSketch, unfortunately, has an initially high memory consumption due to the extra steps of hashing and classification in the data plane, therefore for some tasks the additional memory cost (and complexity if the necessary operation requires the combination of different hashing, classification and counting techniques) is unnecessary in the presence of simpler, more efficient solutions. The hashing and classification steps could also have been better supported if P4 had been used, in that case specific attributes of packets could have been targeted by the rules added/modified by the controller.

This work is the closest to our solution, and although it possesses many of the same properties as UnivMon. It does however perform per-flow fine-grained measurements while losing its hardware independence and its ability to potentially analyze all necessary network protocol.

4.3 Space-Saving Algorithms

Space-Code Bloom Filters (SCBF) [34] are a variation of bloom filters for per-flow traffic measurement. They focus on achieving reasonable measurement accuracy with very low storage and computational complexity.

SCBF represents a multiset, extending the capabilities of traditional bloom filters. Like bloom filters, SCBFs can check if an element is in the multiset but it can also count the number of occurrences of that element. A SCBF uses multiple groups of independent hash functions, each of which can be viewed as a traditional bloom filter. When inserting an element into the SCBF, one of the groups is selected at random and the bits obtained from hashing the element with the different hash functions are set to 1. While querying the SCBF, we count the number of groups where all bits for an element are set to 1. Based on the

number of groups obtained, called observation, we can estimate the multiplicity of the element and return it as the result. However, the existence of heavy-tailed distributions, which are distributions that have heavier tails than the exponential distribution, implies a high multiplicity of elements. Since the group is chosen at random during the insertion operation it is possible that all groups will have the element inserted into them. This unfortunately means that if more elements were to be added to the SCBF, there would be no difference if all groups were already filled. The solution to this problem is the multiresolution SCBF (MRSCBF), in which they employ multiple SCBFs to cover the entire range of multiplicities. For MRSCBFs, when a packet arrives it will be inserted into each SCBF with certain sampling probabilities. SCBFs with high sampling probabilities will keep track of smaller flows and vice-versa. The querying operation is similar to regular SCBFs, however the final estimate is obtained from the estimates of the most relevant SCBFs. One method for estimating the result is the maximum-likelihood estimation (MLE). MLE can be thought of as finding the most likely multiplicity of an element that would have caused the observation of that element. The other method used is mean value estimation (MVE), which computes the multiplicity which is expected to have caused the observation on average.

SCBFs require an increased usage of storage space that is not required for sketches as instead of counting each instance of an element in a multiset (represented by the SCBF) simply employs counters in the structure itself.

The Count-Min Sketch [16] (CM Sketch) is a sublinear space data structure that allows the summarizing of data streams, or sets of packets. CM Sketch allows fundamental queries in data stream summarization such as point, which returns individual items, range, which returns the sum of a sequence of items, and inner product queries, which returns the sum of the product of sequences of items of different sets, to be approximately answered very quickly.

It can be applied to solve several important problems in data streams such as finding quantiles and frequent items for example. The time and space bounds used for the CM sketch to solve these problems significantly improves those previously known, typically from $1/\epsilon^2$ to $1/\epsilon$ in factor. A CM Sketch with parameters is represented by a two-dimensional array counters with width w and depth d, which given parameters (ϵ,δ), makes w=$[e/\epsilon]$ and d=$[ln(1/\delta)]$. Each entry of the array is initially zero. Additionally, d hash functions are chosen uniformly at random from a pairwise-independent family. This sketch shows itself to be exactly what we require, because not only does it utilize sublinear storage space but also as a sketch it is optimized towards frequency counting, and therefore it is difficult to imagine a more efficient data structure at the present.

For our purposes, we will perform mostly point queries, which are performed by identifying the smaller entry for the specific item.

4.4 Applied Network Traffic Measurement

In this section we discuss works that utilize network traffic measurements to perform more complex tasks like DDoS detection or providing quality of service.

Sonata [35] is a system for performing network monitoring line-rate queries at switches for tasks like anomaly detection, real-time application performance analysis and distributed port scan detection. It does so through streaming analytics, which is a method for performing real-time computations on data streaming from applications. The addition of scalable stream processors, devices which exploit parallel processing to perform a great number of operations in a short amount of time, to the network allow Sonata to fuse data streams to answer more sophisticated queries about the network in real-time.

Sonata, like Narayana et al. [29] study, allows users to specify queries to perform measurements. However, they do not define their own hardware primitives for their measurements and instead generates a set of rules from the queries to install. Additionally, while Narayana et al. was capable of aggregating packets, Sonata is able to better refine their measurements.

In the paper, Sonata was implemented with switches running OpenFlow. It uses a runtime system that compiles queries to generate a set of rules to install in the switch data plane and processing pipelines at the stream processor. Additionally, the data-plane operations ensure that traffic filtering is based on relative sampling rates for different flows and that the rate of the filtered data is always less than the system-defined constraints. By abstracting the packet headers and payload as tuples, Sonata facilitates the writing of queries since it allows users to address each packet tuple as a collection of fields, like source and destination address, bytes and ports. Sonata allows users to specify whether a specific operation should be performed at the switch or at the stream processor therefore reducing the workload on them. Finally, Sonata allows users to refine queries based on dynamic conditions. For example, if two different queries are being performed concurrently, then one could sample the entire traffic at a lower sampling rate while the other could sample only the traffic selected by the first query at a higher rate. Query partitioning and refinement allows Sonata to immensely reduce the number of counters and data rate required to efficiently perform measurements.

Although it makes use of SDN technologies and the query language is expressive enough to support our filtering operation, it suffers from the same limitations as OpenNetMon. Due to OpenFlow's limited features it is not possible to perform our specific measurements with Sonata. In order to do so we would have to modify both the OpenFlow implementation in use and the switches function, so that we would be able to query the switch for our metrics. Additionally, the use of sampling could result in the misrepresentation of certain traffic flows.

While this is an elegant and scalable solution to reducing the traffic to be analyzed to a small subset of what passes through the switch it relies on sampling to reduce the amount of traffic it has to inspect. Therefore, making it unfit to what we require, which would require the measuring all traffic.

Sonata can perform fine-grained measurements, does not require packet capture and is easily integrable into networks.

Huang et al. [36] study on the identification of applications aims at improving Quality of Service (QoS) in SDN environments using Machine Learning and DNS Responses. They use a hybrid approach of DNS response identification and ML in order to maximize their accuracy.

Similar to OpenSample [26], Huang et al. captures packets in order to obtain flow attributes. They do so to generate their ML model. However, unlike OpenSample, Huang et al. do not collect only TCP traffic. Additionally, while OpenSample only captures a portion of traffic depending on the sampling rate, Huang et al. do not use sampling.

For the DNS-based method IP addresses and domain names are collected from DNS responses during the training phase. Afterwards, in the classification phase, they check whether source or destination IP address of a flow matches any IP linked to an application name. For ML however, a port mirrors traffic to a traffic classifier in order to obtain flow attributes, like number of packets, packet sizes, throughput and others. After the attributes of a flow are calculated, it obtains a label for the flow a mapping of attributes and a label, which are then used to identify the corresponding applications. Overall, their system works in two phases: First, a training phase, in which the system generates data required for the classification, by either training the ML model or collecting IP address mappings for the DNS-based method. And second, a classification phase, during which either the DNS-based method or supervised ML will be used based on whether or not some requirement is met.

It is possible to perform our measurements by mirroring all traffic towards the classifier and collecting the metrics there and it is even possible to perform filtering through the addition of rules to the SDN switch. However, this requires a heavy use of bandwidth to send all the required packets to classifier and a high amount of memory to keep all the metrics as it does not employ nay space-saving algorithms. Additionally, although it is possible to perform filtering, it can only be done for the protocols supported by OpenFlow and the switch.

The problem with this, however, is how both methods obtain the packets necessary for their operation. Whether it be the DNS-based method or the ML-based method, it is necessary to capture packets for proper classification. This incurs heavy bandwidth usage as entire portions of flows may need to be cloned somewhere other than their original destination and a requires a high amount storage space for storing the necessary packets.

Unlike Sonata [35], Huang et al. do not require sampling but do require packet capture.

Athena [37] is a SDN-based framework for scalable anomaly detection. It intends to support sophisticated anomaly detection services across the entire network data plane while paying special focus to usability, scalability and overhead.

Athena offers API's to users so that they can select the features Athena should collect similar to a query language. Like Sonata [35], the features collected are turned into SDN rules that are installed into the switch and are limited by OpenFlow's capabilities. Additionally, Athena measurements do not require

sampling, although sampling rates can be specified. Athena queries can also aggregate information similar to Narayana et al. [29] study but do not possess the same level of refinement as Sonata.

The Athena system is composed of three major elements: the Athena Southbound (SB) Element instances, which isolates control messages, extracts features to drive the analysis algorithms and mitigates detected problems, run above multiple parallel controllers for scalability and allows for monitoring and issuing of control messages. The Athena Northbound (NB) Element, which exports Northbound APIs for application developers to utilize Athena's functionalities for anomaly detection for scalability and transparency. The Athena Off-The-Shelf Strategies, which include preset anomaly detection strategies. The features required for these strategies belong to one of these three categories: Protocol-centric features, which are directly derived from control messages and include packet and byte counts. Combination features, which are derived from predefined formulas using protocol-centric features and include flow packet counts and duration. And Stateful features, which are the states of the network operations and include the pair flow ration. In real-time, users can use Northbound API's to specify to Athena what features to collect from traffic and what kind of anomaly detection ML model to train using the collected features. These ML models can then be used to detect anomalies from future collected features.

It is not possible to perform our measurements on Athena without modifying OpenFlow's features and the switch's functioning. This is due to OpenFlow's features not being modifiable to collect packet length distributions like we require. While Athena does use SDN, allowing it to manage multiple network devices simultaneously, it does not use space-saving algorithms to reduce the amount of space used. Additionally, due to OpenFlow's fixed-function, it cannot operate over all protocols and hardware.

Unfortunately, the features gathered by the switches, which are then sent to the Athena instances through control messages, are stored without the use of any space-saving algorithms the storage space used may raise to worrying levels.

Similar to the previous two works, it provides per-flow fine-grained measurements but lacks the necessity for sampling or packet capture.

4.5 Discussion

Among all the works in the previous sections, only a select few utilized space-saving algorithms and even fewer chose to use counting sketches. And as we suggested for many of those that did not use them, sketches could provide a much more efficient way to store metrics. In the case of the ones that utilized sampling, the use of sketches could have made the choice to do just that unnecessary. Counting sketches require only a sublinear amount of storage space and are optimized towards counting operations, and, in some cases, making so that what could potentially be terabytes of data be represented in just kilobytes instead. In some cases, it could even be more efficient than using sampling at all (depending on the sampling ratio) and, as is the case of Duffield et al. study [30], could bypass having to perform some operations that would incur additional complexity.

Table 1. Comparison of properties of works discussed to our solution. (PFM:Per-Flow Fine-Grained Measurement, PC:Packet Capture, S:Sampling, SSA:Space-Saving Algorithms, EN:Existing Networks, DSPH:Dependent on Specific Protocols or Hardware)

	PFM	Does Not Use PC	Does Not Use S	Uses SSA	Integrable into EN (via SDN)	Not DSPH (viaProg. Switches/P4)
Duffield et al. [30]	✗	✓	✗	✗	✗	✗
OpenSample [26]	✗	✗	✗	✗	✓	✗
OpenNetMon [28]	✓	✓	✗	✗	✓	✗
Sonata [35]	✓	✓	✗	✗	✓	✗
Huang et al. [36]	✓	✗	✓	✗	✓	✗
Narayana et al. [29]	✓	✓	✓	✗	✗	✓
Athena [37]	✓	✓	✓	✗	✓	✗
FlowRadar [31]	✓	✓	✓	✓	✗	✗
UnivMon [32]	✗	✓	✓	✓	✓	✓
OpenSketch [33]	✓	✓	✓	✓	✓	✗
Our Solution	✓	✓	✓	✓	✓	✓

Additionally, in FlowRadar [31] we saw that besides the flow counters, they also encoded the flows themselves into their data structures so that network wide decoding could be employed. However, we argue that by employing a system similar to FlowRadar in a SDN environment, there would be no need for doing such a thing, as the dynamic control it provides us over the data plane, it would be trivial to match each flow to their counters. Either by simply keeping track of some identifier for each specific flow in the controller, or if P4 is being used, tagging each packet of each different flow with a unique identifier and simply matching them across network devices.

Table 1 summarizes and compares the works we discussed in the previous sections to our solution according to the six properties mentioned earlier.

5 Architecture

We plan to build a system that allows for the gathering of metrics in a SDN environment. It uses programmable switches and P4 to implement the Count-Min sketch over the packet processing pipeline to efficiently store the metrics as estimates and allow the filtering of packets over their individual network protocols and header fields.

It can track all flows passing through the switch while storing measurements that allow for the storage of estimated metrics in sublinear space. Our system will, for each flow, keep store in the sketch "buckets" for different packet lengths in increments of N and increment one of them for each packet belonging to that flow it sees. From there, every time the controller requests the current sketch, it will be able to compute a series of metrics for timeseries of packet lengths and packet inter-arrival times, including the minimum, maximum and skew for example. Finally, an API should allow users to limit the traffic to be analyzed through the addition of filtering rules, which are first sent to the controller

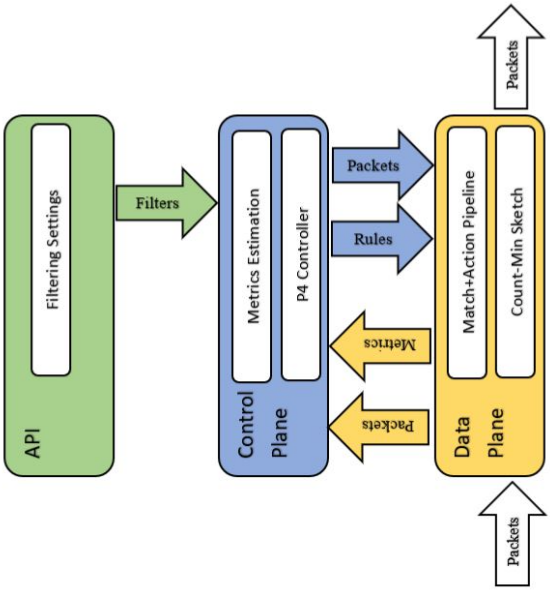

Fig. 7. Brief overview of the system.

and then propagated to the switch. Overall, the measurement operation should attempt to record the measurements while incurring the least amount of overhead as possible. Figure 7 shows a diagram of our architecture.

Compared to OpenSketch [33], one of the previous works discussed that is the most comparable to our own, we can highlight several differences. First, the hashing and classification of OpenSketch can be made more efficient through the matching tables used in P4. Second, while OpenSketch can perform filtering to a certain degree our own filtering can be altered at runtime to suit the user's needs and to a higher degree of detail. This is because P4 can support any network protocol necessary and selectively target any header fields present in them. Finally, OpenSketch's data plane implementation may need to be adapted to the hardware its running on while ours in P4 is hardware-independent and will always be the same.

In the following subsections, we discuss how our implementation stores the metrics and how the metrics are interpreted at the controller and estimate the wanted metrics from that.

5.1 Metrics Storage

In this section, we present a basic description of the operation of the switch and data plane and how they store the required metrics from incoming packets. Starting by checking if the packets belong to a preexisting flow or not, and adding

a rule for it if not, how the sketch allows us to save space and how we store the metrics within it.

Rule Addition To avoid excessive interaction between the switch's data plane and the SDN controller we will preprogram a few permanent rules into the switch focused around protocols of layers 2 to 4 of the OSI layer and the few machines we will connect to the switch. These rules will focus on matching incoming packets by their MAC addresses, IP addresses and port numbers (when applicable) and may perform one of two actions. Either they forward the packet towards the correct destination or they drop the packet if it matches no rules. The controller should also keep a list of all active rules so that when the controller receives the sketch it can identify which entries belong to which flows.

There is another method of adding rules to the switch. Similar to Figure 2, if a packet has no match in the switch it would be cloned towards the SDN controller through a special "CPU port". There the controller would retrieve relevant fields to uniquely identify the flow and to identify where it should be forwarded to and install it in the switch. Afterwards, the controller will resend the packet towards the switch where it will be matched to the new rule and be forwarded towards the correct destination.

Filtering Before adding a packet's characteristics to the counting algorithm, it must first be decided if that should be counted at all or not. This filtering operation will serve to limit the analyzed traffic to a smaller subset than the total traffic passing through the switch based on the types of protocols used or header field information. The filtering operation will be implemented over one of P4's match+action tables, which will include one or more rules for the packets to be matched to. As seen in Figure 3, the tables are primarily made up of keys and actions. The keys are the fields, whether they be from meta data or from headers, to be filtered. Examples of these keys may include protocol headers, header flags and IP or MAC addresses. The actions will either simply change a flag to signify that that packet will be counted or not do anything. Afterwards, before the counting operation, the flag is checked and if it is indeed set, then the packet will be counted in the following operation.

Sketch Although an experimental analysis is still required, we can provide an example of Count-Min Sketch's [16] use and its efficiency. We choose an error $\epsilon = 0.01$ and a probability$\delta = 0.01$, or $w = 272$ and $d = 5$ which would produce an array of size approximately 1360 entries. Additionally, to determine the amount of bits per entry we would need to add the maximum amount of all counters we would need to maintain, α, and obtain the result by performing $\log(\alpha)$. Assuming a rough estimate of 500000 counters necessary (this can be calculated by multiplying the number of expected concurrent flows with the number of "buckets" necessary to draw a proper distribution of packet lengths, in this case the "buckets" are in increments of 5) that can go up to the maximum

number in a 32-bit value that would leave us with approximately 16 bits per entry. Which leads to a total of approximately 22 kilobits.

Counting The CM Sketch is implemented underneath the match+action pipeline of data plane. After the filtering operation, the P4 program will check if a meta-data flag is set, signifying that that packet may be counted in the sketch. This is performed through a fixed rule in a P4 table that will either immediately resume the forwarding of the packet, if the flag is not set, or perform an action that enables the sketch insertion operation. The sketch is meant to track the distribution of packet lengths for each flow, as such there will be one entry in each row of the sketch for each "bucket" of the distribution for each individual flow. In order to facilitate this, we hash (with the hash functions created for the sketch) several key header fields and the packet length to select the entry counters that packet should increment, before allowing the packet to continue on its way. This allows the measurement operation to be performed in a mostly pass-through fashion minimizing the added overhead as much as possible.

5.2 Metrics Calculation

In this section, we present a basic description of the operation of the controller and control plane. Mainly, how the controller can obtain the sketch from the switch and compute the necessary metrics.

Adding to the Filtering Operation Through the P4Runtime API it is possible to both add, delete and modify rules, or table entries as they are referred to in the API, to the filtering table. Due to the way P4 tables work, it is only possible to add rules that contain either meta data or protocol header fields that are already supported be the P4 program. Therefore, if the user wants to filter for fields not currently supported by the program, the user needs to first add them to the P4 program.

Reading the Sketch By utilizing a slightly modified version of the P4Runtime API, the sketch can be serialized at the switch and retrieved at the controller. A reading for the current state of the sketch should be performed periodically, in a similar manner to how the value of counters is read. Additionally, in order to limit the danger of overflowing a sketch entry, and to possibly decrease the number of bits per entry, the sketch should be zeroed after reading the values stored within it. The frequency with which we read the sketch should be defined by a user variable.

Calculating the Metrics First, we can attain the packet length distributions for all flows by simply calculating each minimum value for each "bucket" present in the sketch, since it keeps multiple entries for each "bucket". Next, we can also add all the ones belonging to the same flow, allowing us to calculate both the

average packet length, by multiplying the size assigned to each "bucket" to the counted frequency and dividing that by the sum of all frequencies for that flow, and the average inter-arrival time, by dividing the time between each reading of the sketch by the sum all the frequencies for the "bucket" of that flow. Both the average packet size and inter-arrival time per-query is stored as a different timeseries. As new timeseries are added, we can recalculate the maximum, minimum, mean, percentiles, skew and kurtosis of the previous timeseries with the newer ones.

5.3 Filtering API

In this section, we present a basic description of the operation of the filtering API. Mainly, how the terminology is used to add new filtering rules to the switch.

Terminology When adding keys to a P4 table entry, both the name of the field and the value the user wishes for it to be matched to need to be specified. Therefore, new filtering table entries may be quickly added by matching P4's table terminology to the API's. Specifically, when selecting fields for tables only a string with that fields name is necessary. The fields are divided to either header fields, which contain the word "hdr", and meta data fields, which contain the word "meta". Furthermore, "hdr" fields must be contain the name or abbreviation of their corresponding protocol, like "ipv4" or "tcp". Finally, for both meta data and header fields, the actual name of the comes last. Examples of this can be found in Figure 3, which uses both the destination address of the ipv4 protocol and a virtual id of a custom network protocol header called meta.

6 Evaluation

6.1 Base Evaluation

In order to evaluate our proposal, we will implement our system over a Barefoot Tofino programmable switch with 64 100G ports and capable of 6.4Tbps throughput. For thorough testing, several traffic traces must be attained and then transmitted through our switch, preferably towards a local machine so that we can confirm the correct forwarding of the packets, and with different filtering conditions for different tests. At the same time, we will have a separate machine to monitor the switch for some of the evaluation metrics we will require. The metrics we chose to evaluate our system are:

- The latency of processing packets. This will determine the extra overhead added by the measurement operation to the switch's regular operation by comparing it to the switch's operation without it.
- CPU usage of the switch. This will determine if the overhead added by our measuring operation is stressing the switch by comparing it to the switch's operation without it.

– Memory compression of the metrics. This will determine the amount of space saved by our chosen space-saving algorithm over other alternatives.
– Accuracy of the metrics. This will determine the proximity of the estimated metrics stored by our chosen space-saving algorithm to reality.

The first three metrics are to be tracked by an additional machine monitoring the switch querying it for its throughput, CPU usage and storage space used. The machine will keep track of both peaks and average values of these metrics for comparison between the switch's performance with and without the measurement operation using terminal commands for the switch. The final metric will be evaluated by comparing the attained packet length distributions and packet length and inter-arrival time timeseries statistics with the real distribution and statistics attained from the same traces through the use of functions that will allows us to compare them. Using metrics like the Kolmogorov-Smirnov distance, Kullback-Leibler divergence, or even the Earth-Mover's Distance as seen in Barradas et al. [9] study.

6.2 Applied Evaluation

As a final test, we will attempt to direct our system towards the purpose of identifying Tor traffic and covert Skype traffic identification as described in Barradas et al [9] study. The study highlights the use of several different Decision Trees, in which a model in the form of a tree, where each tree node is either a decision or a label and the nodes closer to the root have a higher importance, Random Forests, where labels are decided by a majority vote from the output of multiple trees trained by random samples, and eXtreme Gradient Boosting (XGBoost), which begins with a shallow decision tree but that each step of the algorithm optimizes the predictions made in earlier stages.

The main evaluation metrics to track in this test are the storage compression and accuracy of the metrics as the primary challenge will require us to experiment with and carefully deliberate upon the most apt accuracy/storage tradeoff for the sketch. This tradeoff will impact the success rate for the correct identification of flows corresponding to covert Skype traffic as well as the efficiency with which it does so. For this test, a similar set up to the previous test should be used but this time using covert Skype traffic traces should be mixed in with regular traffic in order to properly test the system's ability.

7 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

8 Conclusions

For years, network traffic measurement has provided network analysts and technicians with the tools and technology to efficiently maintain and optimize their networks through several measurement tasks like the ones discussed in Callado et al. [1] and Mohan et al. [2] surveys. The introduction of modern networking advancements, specifically SDNs, programmable switches and P4, have allowed for never before seen levels of efficiency and accuracy through simple to implement, and extend, mechanisms.

While several works and techniques have been introduced to take advantage of these advancements, some of the limitations of older tools are still present in these implementations, most often suffering from storage space limitations and having to resort to alternative solution that require sampling or additional hardware. And even then, those that have surpassed those limitations and drawbacks still do not provide the networks with measurements that possess the level of detail, accuracy and efficiency that can be drawn out from the right combination of SDNs, programmable switches, P4 and counting sketches, which are key for overcoming the previously mentioned limitation.

We propose a system that takes advantage of all of these techniques, which builds on previous, and just as impressive, works, inspired by the likes of OpenSketch [33] and UnivMon [32], that provides line-speed, efficient fine-grained characterizations of individual flows while allowing analysts and technicians to carefully adjust the tradeoff between metric accuracy and storage space used as well as filter the traffic they want to analyze by the properties of their headers.

Acknowledgments We are grateful to Diogo Barradas for fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) and Feder through the project UID/ CEC/ 50021/ 2019.

References

1. Callado, A., Kamienski, C., Szabó, G., Gero, B.P., Kelner, J., Fernandes, S., Sadok, D.: A survey on internet traffic identification. IEEE Communications Surveys & Tutorials **11**(3) (August 2009) 37–52
2. Mohan, V., Reddy, Y.R.J., Kalpana, K.: Active and passive network measurements: A survey. International Journal of Computer Science and Information Technologies **2**(4) (2011) 1372–1385
3. Netflow. https://www.ietf.org/rfc/rfc3954.txt Accessed: 2018-12-31.
4. Ping, Traceroute. https://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-software-releases-121-mainline/12778-ping-traceroute.pdf Accessed: 2018-12-31.
5. Wireshark. https://www.wireshark.org/docs/wsug_html_chunked/ Accessed: 2018-12-31.
6. SNMP. http://www.net-snmp.org/docs/readmefiles.html Accessed: 2018-12-31.

7. SDN. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf Accessed: 2018-12-31.
8. Barefoot. https://www.barefootnetworks.com/products/brief-tofino/ Accessed: 2018-12-31.
9. Barradas, D., Santos, N., Rodrigues, L.: Effective detection of multimedia protocol tunneling using machine learning. SEC '18 Proceedings of the 27th USENIX Conference on Security Symposium (August 2018) 169–185
10. Hayes, J., Danezis, G.: k-fingerprinting: A robust scalable website fingerprinting technique. In: 25th USENIX Security Symposium, Austin, Texas, USA (August 2016) 1187–1203
11. Taylor, V.F., Spolaor, R., Conti, M., Martinovic, I.: Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrucken, Germany (March 2016)
12. Schuster, R., Shmatikov, V., Tromer, E.: Beauty and the burst: Remote identification of encrypted video streams. In: 26th USENIX Security Symposium, Vancouver, Canada (April 2017)
13. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review **44**(3) (July 2014) 87–95
14. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7) (July 1970) 422–426
15. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. ICALP '02 Proceedings of the 29th International Colloquium on Automata, Languages and Programming (July 2002) 693–703
16. Cormodea, G., Muthukrishnan, S.: An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms **55**(1) (April 2005) 58–75
17. Zhang, Y., Singh, S., Sen, S., Duffield, N., Lund, C.: Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. IMC '04 Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (October 2004) 101–114
18. Carl, G., Kesidis, G., Brooks, R.R., Rai, S.: Denial-of-service attack-detection techniques. IEEE Internet Computing **10**(1) (January 2006) 82–89
19. Liao, H.J., Lin, C.H.R., Lin, Y.C., Tung, K.Y.: Intrusion detection system: A comprehensive review. Journal of Network and Computer Applications **36**(1) (January 2013) 16–24
20. Fluke. https://assets.tequipment.net/assets/1/26/Fluke_Tap_Solution_Network_Monitoring_and_Analysis_Techniques_Using_Taps_and_SPAN_Switches.pdf Accessed: 2018-12-31.
21. Ahmed, N.K., Neville, J., Kompella, R.: Network sampling: From static to streaming graphs. ACM Transactions on Knowledge Discovery from Data (TKDD) **8**(2) (June 2014) 7:1–7:56
22. Zhang, M., Dusi, M., John, W., Chen, C.: Analysis of udp traffic usage on internet backbone links. In: 2009 Ninth Annual International Symposium on Applications and the Internet, Bellevue, Washington, USA (July 2009)
23. John, W., Tafvelin, S.: Analysis of internet backbone traffic and header anomalies observed. IMC '07 Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (October 2007) 111–116

24. Murray, D., Koziniec, T.: The state of enterprise network traffic in 2012. In: 2012 18th Asia-Pacific Conference on Communications (APCC), Jeju Island, South Korea (October 2012)
25. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review **38**(2) (April 2008) 69–74
26. Suh, J., Kwon, T.T., Dixon, C., Felter, W., Carter, J.: Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In: 2014 IEEE 34th International Conference on Distributed Computing Systems, Madrid, Spain (July 2014)
27. sFlow. http://sflow.org/about/index.php Accessed: 2018-12-31.
28. van Adrichem, N.L.M., Doerr, C., Kuipers, F.A.: Opennetmon: Network monitoring in openflow software-defined networks. In: 2014 IEEE Network Operations and Management Symposium (NOMS), Krakow, Poland (May 2014)
29. Narayana, S., Sivaraman, A., Nathan, V., Alizadeh, M., Walker, D., Rexford, J., Jeyakumar, V., Kim, C.: Hardware-software co-design for network performance measurement. HotNets '16 Proceedings of the 15th ACM Workshop on Hot Topics in Networks (November 2016) 190–196
30. Duffield, N., Lund, C., Thorup, M.: Estimating flow distributions from sampled flow statistics. SIGCOMM '03 Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (August 2003) 325–336
31. Li, Y., Miao, R., Kim, C., Yu, M.: Flowradar: A better netflow for data centers. NSDI '16 Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI) (March 2016) 311–324
32. Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., Braverman, V.: One sketch to rule them all: Rethinking network flow monitoring with univmon. SIGCOMM '16 Proceedings of the 2016 ACM SIGCOMM Conference (August 2016) 101–114
33. Yu, M., Jose, L., Miao, R.: Software defined traffic measurement with opensketch. NSDI '13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI) (April 2013) 29–42
34. Kumar, A., Xu, J., Wang, J.: Space-code bloom filter for efficient per-flow traffic measurement. IEEE Journal on Selected Areas in Communications **24**(12) (November 2006) 2327–2339
35. Gupta, A., Birkner, R., Canini, M., Feamster, N., Mac-Stoker, C., Willinger, W.: Network monitoring as a streaming analytics problem. HotNets '16 Proceedings of the 15th ACM Workshop on Hot Topics in Networks (November 2016) 106–112
36. Huang, N.F., Li, C.C., Li, C.H., Chen, C.C., Chen, C.H., Hsu, I.H.: Application identification system for sdn qos based on machine learning and dns responses. In: 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), Seoul, South Korea (September 2017)
37. Lee, S., Kim, J., Shin, S., Porras, P., Yegneswaran, V.: Athena: A framework for scalable anomaly detection in software-defined networks. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, Colorado, USA (June 2017)