Detecting Heavy Flows in the SDN Match and Action Model

Yehuda Afek[a], Anat Bremler-Barr[b], Shir Landau Feibish[a], Liron Schiff[a]

[a] *Blavatnik School of Computer Science, Tel-Aviv University, Israel*
[b] *Computer Science Dept., Interdisciplinary Center, Herzliya, Israel*

Abstract

Efficient algorithms and techniques to detect and identify large flows in a high throughput traffic stream in the SDN match-and-action model are presented. This is in contrast to previous work that either deviated from the match and action model by requiring additional switch level capabilities or did not exploit the SDN data plane. Our construction has two parts; (a) new methods to efficiently sample in an SDN match and action model, (b) new and efficient algorithms to detect large flows efficiently and in a scalable way, in the SDN model.

Our large flow detection methods provide high accuracy and present a good and practical tradeoff between switch - controller traffic, and the number of entries required in the switch flow table. Based on different parameters, we differentiate between heavy flows, elephant flows and bulky flows and present efficient algorithms to detect flows of the different types.

Additionally, as part of our heavy flow detection scheme, we present sampling methods to sample packets with arbitrary probability p per packet or per byte that traverses an SDN switch.

Finally, we show how our algorithms can be adapted to a distributed monitoring SDN setting with multiple switches, and easily scale with the number of monitoring switches.

Email addresses: afek@cs.tau.ac.il (Yehuda Afek), bremler@idc.ac.il (Anat Bremler-Barr), shirl11@post.tau.ac.il (Shir Landau Feibish), schiffli@post.tau.ac.il (Liron Schiff)

This research was supported by European Research Council (ERC) Starting Grant no. 259085, and the Neptune Consortium, administered by the Office of the Chief Scientist of the Israeli ministry of Industry, Trade, and Labor, and the Ministry of Science and Technology, Israel.

1. Introduction

Heavy flow detection in traffic remains one of the fundamental capabilities required in a network. It is a critical telemetry tool for understanding the behaviour of traffic in a given point in the network or in the network as a whole. Furthermore, it is a key ability in providing QoS, capacity planning and efficient traffic engineering. Additionally, heavy flow detection is crucial for the detection of Distributed Denial of Service (DDoS) attacks in the network which remain a common attack in the Internet today, with hundreds of attacks carried out daily [1].

We present techniques for large flows detection in traffic that flows through a Software Defined Network (SDN) switch. While SDN switches are very efficient and considerably simpler to manage than existing routers and switches, they do not offer direct means for the detection of large flows.

Existing network monitoring tools for classic IP networks have been available for over 20 years, with one of the earliest tools being Cisco Netflow [2]. Over the years, traffic visibility, and specifically measurements and monitoring in IP networks has become an increasingly difficult task due to the overwhelming amounts of traffic and flows [3]. While existing tools may be very useful for classic networks, monitoring in SDN networks requires new tools and technology. The SDN network architecture places the controller as the focal point of the network. Therefore, using existing tools would require extensive communication between the controller and the monitoring tools, which would place significant overhead on the controller. It is therefore necessary to provide new monitoring methods for SDN networks based on the SDN architecture.

We design ways to implement monitoring methods with the widespread OpenFlow and the recent P4 standard for SDN switches. OpenFlow switches provide counters of the number of bytes and packets per flow entry, yet traffic measurement remains a difficult task in SDN for two reasons: First the hardware (usually Ternary Content Addressable Memories (TCAMs)) constraints limit the number of flows which the switch can maintain and follow. Secondly the limited number of updates that the switch can process per second [4], which hence limits the number of updates that the controller can make to the flow table. The algorithms provided herein overcome these limitations by providing efficient building blocks for large flow detection and sampling which may be used by various monitoring applications.

1.1. Our Contribution

First, we propose our Sample&Pick algorithm which is an efficient method to detect large or heavy flows going through an SDN switch. The Sample&Pick algorithm is designed for protocols which are based on the match and action model

(e.g., OpenFlow, *P*4, etc.), and performs a division of labour between the switch and the controller, coordinating between them to identify the large flows. Sample&Pick achieves very high accuracy using a fixed amount of rules in the switch and requiring little communication between the switch and the controller.

Second, as part of our algorithm we present various OpenFlow (with optional features) based methods to sample packets that traverse an SDN switch. These methods may be used independently of our heavy flows detection algorithm.

Third, we consider a distributed model with multiple switches and propose solutions for efficient scaling of our techniques, to support large flow detection as well as sampling in the distributed setting.

Finally we have implemented and evaluated our Sample&Pick comparing it with OpenSketch [5]. The sampling methods rely on standard and optional features of OpenFlow 1.3 (or the *P*4 language) and are implemented with the NoviKit (hardware) switch[6] (operated with NoviWare switching software [7]) that supports the optional OpenFlow feature to match on extra fields. The heavy flow detection also relies on a standard OpenFlow controller and was evaluated as a whole using a dedicated virtual time simulation for both the data and control planes. Additionally, the techniques presented are both flow-table size and switch-controller communication efficient.

2. Preliminaries

2.1. Background: Heavy Hitters

The *Heavy Hitters Problem* (also known as the ε-Approximate Frequent Items Problem) is defined as follows: Given a sequence of N values $\alpha = \langle \alpha_1, ...\alpha_N \rangle$, a threshold $0 \leq \theta \leq 1$ and an error value ε: denote $f_x = |\{j|\alpha_j == x\}|$, find a set of items F such that for any item $x \in \alpha$, if $f_x \geq \theta N$ then $x \in F$, and if $f_x \leq \theta N - \varepsilon N$ then $x \notin F$.

Note that a streaming algorithm for the Heavy Hitters problem, can use only a constant amount of space and make a single pass over the input.

Many solutions have been proposed for the heavy hitters problem, for example [8, 9, 10, 11, 12]. A description of a few counter-based algorithms as well as other results regarding the heavy hitters problem can be found in [13].

We chose the Space Saving algorithm of Metwally et al. [8] as a building block for the detection of heavy flows due to its simplicity, efficiency and high level of counter accuracy.

The Space Saving algorithm works as follows: given parameters ε, θ as described above and a sequence of N values $\alpha = \langle \alpha_1, ...\alpha_N \rangle$, the algorithm maintains $v = \frac{1}{\varepsilon}$ items, each consisting of an ID and a counter. For each value $w = \alpha_i$ for $1 \leq i \leq N$:

1. If the *ID* of one of the v items equals w, its counter is incremented by 1.

2. Otherwise, let u be an item with the minimal counter value. The *ID* of u is replaced with w and its counter is incremented by 1.

The (additive) error rate $N\varepsilon$ of this algorithm for $\varepsilon = \frac{1}{v}$ is $\frac{N}{v}$ [8]. Therefore, for each item j, denote its counter in the output of the algorithm c_j, and its count in the sequence as r_j then $r_j \leq c_j \leq r_j + N\varepsilon$. The algorithm requires $O(v)$ space and only a single pass over the input, with a small number of instructions per item.

2.2. Definitions

Following [14] a *flow* is defined to be any sequence of packets which can be matched to rules in the flow table, such as, for example, those defined by a set of header field values. Note that our algorithms can be used for any flow definition, including those which pertain to matches in the payload or any of the headers as long as it is supported by the controller and switch implementation. A flow entry in an OpenFlow (flow) table can be defined to match packets according to (almost) any selection of header field bits thereby allowing various flow definitions.

A *large flow* is usually defined as a flow that takes up more than a certain percentage of the link traffic during a given time interval [15]. For some applications other definitions of large flows are required, for instance network analysis tools may want to identify flows that consist of a certain amount of packets regardless of link capacity.

We therefore refine the large flow definition, considering both the time aspect as well as the type of measurement performed.

We consider the following definitions of large flows:

Definition 2.1. ***Heavy flow:*** *Given a stream of packets S, a* heavy flow *is a flow which includes more than T percent of the packets since the beginning of the measurement.*

Considering the definition of *flow* provided above, this can be useful for identifying flows which remain heavy over a significant period of time, for example in Distributed Denial of Service (DDoS) attacks. On the other hand this will miss large flows if the measurement continues for a very long period of time.

Definition 2.2. ***Interval Heavy flow (Elephants):*** *Given a stream of packets S, and a length of time m, an* interval heavy flow *is a flow that includes more than T percent of the packets seen in the previous m time units.*

These can be used for standard traffic management and resource allocation. We note that heavy flows in the context of the streaming model are defined with respect to the entire stream as opposed to Interval Heavy flows which are heavy within a given time interval. Algorithms such as the Space Saving algorithm [8] do not offer a direct mechanism for detecting heavy hitters with respect to part of the stream and therefore new techniques are required for the detection of Interval Heavy Flows.

Definition 2.3. ***Bulky flow at a point of time:*** *Given a stream of packets S, and a length of time m, a* bulky flow *is a flow that contains at least B packets in the previous m time units.*

The algorithms we present for large flows follow the above definitions which consider traffic volume measurements in terms of packets. Nevertheless, we note that certain traffic management capabilities require volume, i.e., byte size, analysis. For instance, if we wish to identify the flow which takes up the most bandwidth, then we are required to count the number of bytes in the flow rather than the number of packets. The algorithms presented here work well for both definitions.

2.3. Towards a Solution in SDN

Fundamental counter based algorithms for finding Heavy Hitters (or flows) such as that of Metwally et. al. [8], cannot be directly implemented in the SDN framework since in the worst case they would require rule changes for every packet that traverses the switch. A different approach is therefore needed.

First we consider a naive solution which we name Sample&HH, that samples packets in the switch and then sends all sampled packets to the controller. The controller computes the heavy flows using a heavy hitters algorithm. However, as can be seen in Figure 4a (and other works [15]), relying solely on the samples is not accurate enough. Next we consider a solution based on the *Sample&Hold* paradigm of [15] which was devised for identifying elephant flows in traffic of classic IP networks. In Sample&Hold sampled packets are sent to the controller, which installs a counter rule for each new flow that is sampled. Every consequent packet from that flow will be counted by the rule and will not be sampled. By using sampling together with accurate in-band counters for sampled flows Sample&Hold achieves very accurate results, yet the high amount of counters and the rate of installing them make Sample&Hold incompatible with SDN switch architecture. Therefore we only consider it as a reference point to evaluate our algorithm.

To deal with the problems of the above solutions, we present our Sample&Pick algorithm. Sample&Pick uses sampling to identify flows that are suspicious of being heavy. For these *suspicious* flows a special rule is placed in the switch flow table providing exact counters for them. The Sample&Pick algorithm considers

both the bounded rule space in the switch as well as the time it takes for the controller to install a rule in the switch. Therefore we use two separate thresholds, the first, T, for determining which flows are heavy and a second lower threshold, t, for detecting potentially large flows. This lower threshold allows us to install rules in the switch early enough to get an accurate count of the large flows, yet we do not install rules for too many flows that will remain small. The Sample&Pick algorithm is described in detail in Section 3.1.

Table 1 depicts the conceptual differences and the resource consumption overhead of the Sample&Pick algorithm, the SDN Sample&Hold algorithm and the Sample&HH algorithm.

Technique	Switch memory usage	Controller functionality	Controller to Switch traffic	Switch to controller traffic
Sample&Pick	Sampling rules + at most $\frac{1}{t}$ count rules	Heavy hitters computation + counter aggregation	Every interval at most $\frac{1}{t}$ new count rules	Sample of all non-hold packets + counters each interval.
Sample&Hold (OpenFlow variant)	Sampling rules + *unlimited* count rules	Counter aggregation	Every new sample create message with a new count rule	Sample of all non-hold packets + final counters.
Sample&HH	Sampling rules	Heavy hitters computation	None	Sample of *all* packets

Table 1: Comparison of the heavy flow detection techniques presented in this paper. Denote t the threshold for candidate heavy hitters in Sample&Pick.

3. Heavy Flows Detection

3.1. The Sample&Pick algorithm

3.1.1. Algorithm Overview

Our algorithm operates as follows: in the first step we sample the flows going through the switch using one of the sampling techniques to be explained in Section 4. As can be seen in Fig. 1, these samples are sent to the controller, that feeds them as input to a heavy hitters computation module in order to identify the

name	match	actions
Count $flow_1$	$(src\_ip, src\_port, dst\_ip, dst\_port) = flow_1$	1
...	...	...
Count $flow_m$	$(src\_ip, src\_port, dst\_ip, dst\_port) = flow_m$	1
Sample	$(src\_ip, src\_port, dst\_ip, dst\_port) = *$	2

Table 2: Illustration of switch flow table configuration. Rule priority decreases from top to bottom. Actions: 1- increment counter; 2 - apply sampling technique (goto sampling tables / apply group)

suspicious heavy flows (steps 2 and 3). Once a flow's counter in the heavy hitters module has passed some predefined threshold t, a rule is inserted in the switch to maintain an exact packet counter for that flow (steps 4 and 5). This counter is polled by the controller at fixed intervals and stored in the controller (steps 6 and 7). Finally the last step increments the counters that are processed by the Heavy Hitters module to maintain correct counters of non-sampled flows.

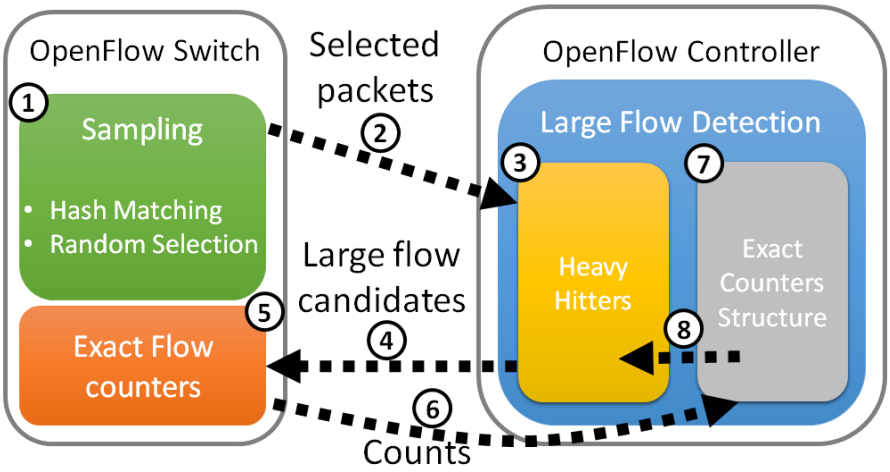


Figure 1: Sample&Pick overview

3.1.2. Switch Components Design

As seen in Fig. 1, two kinds of rules are used in the switch flow tables. The *sampling* rules, which are created as needed by the selected sampling algorithm as described in Section 4. And the *counter* rules used for precisely counting packets of potentially heavy flows. An example of this configuration can be seen in Table 2.

First, each packet is matched against *counter* rules. In case of a successful match, the relevant counter is increased. Only if the packet does not match any

counter rule, it is matched against the sampling rules, and if the packet is selected by the sampling rules it (or only the headers) is sent to the controller. Counters of the *counter* rules are only sent to the controller when polled by the controller.

3.1.3. Controller Components Design

As shown in Fig. 1, the controller maintains the heavy hitters computation module and a collection of the exact counters accumulation.

The *heavy hitters computation module*: Maintains the heavy hitters data structure according to the algorithm of Metwally et al. [8], as described in Section 2.1.

As the heavy hitters module only receives sampled data which is sent to the controller from the switch, the traffic of the heavy flows which are not sampled is not inserted at all into the heavy hitters and therefore it may seem as though the flows are no longer heavy. To simulate the sampling of these heavy flows, when the controller polls the switch for the updated counters, it uses those counters to update the heavy hitters module accordingly. That is, we simulate a sampling of the heavy flows by updating the heavy hitters module with the number of new packets that have been counted since the previous polling, multiplied by the sampling ratio p. As noted, this mechanism saves a substantial amount of sample traffic from the switch to the controller.

The *exact count data structure*: The accumulated counters of the flows that are suspected to be heavy are maintained in a simple ordered data structure. Its use is to compute the delta from the previous time the counters were polled. This delta is then fed (with a factor) into the heavy hitter module.

An additional counter is maintained in the controller to count the total number of items inserted into the heavy hitters module, which is necessary to calculate the rates from the individual counters inside the heavy hitter module. At any point the heavy flows may be identified as the flows in the heavy hitters module that have passed the threshold T, relative to the total counter.

3.1.4. Analysis

Here we discuss how to choose the parameters, t and v of Sample&Pick algorithm for given problem parameters, the threshold T for heavy flow and the sampling probability p. Recall that v is the number of items maintained by the Space Saving algorithm used in the controller component (see Section 2.1). The threshold t is used to detect large flow candidates (see Section 3.1). By definition, if a total of N packets have passed so far, each heavy hitter flow contains at least TN packets. Our controller receives each packet with probability p. The number of samples is then on average (or exactly depending on the sampling method) $n := Np$. The number of packets sampled out of x original packets is a random binomial variable with average xp and variance $xp(1-p)$. When x is high this

converges to normal distribution with similar parameters. For normal distribution, w.h.p the random variable is within distance of 3 times the standard deviation from the average. Therefore the number of packets sampled from x packets is w.h.p greater than $xp - 3\sqrt{xp(1-p)}$.

Our scheme uses a threshold $t < T$, in order to detect possible heavy flows that might be missed due to sampling errors. For a heavy flow (with at least $T \cdot N$ packets) w.h.p at least $TNp - 3\sqrt{TNp(1-p)}$ packets are sampled. We need to set t to ensure that the above expression is higher than $t \cdot n$. Thus,

$$t < T - 3\frac{\sqrt{T(1-p)}}{\sqrt{Np}} \tag{1}$$

Since t must be a positive number, we get the following constraint on the flow weight (ratio) our scheme is expected to detect: $T^2 - 9\frac{T(1-p)}{Np} > 0$ which is valid when

$$T > 9\frac{1-p}{Np} \tag{2}$$

To illustrate the relationship between the different parameters, we examine the following example: Given a line rate of $6 \cdot 10^5$ packets per second and a controller throughput of only a few thousands messages per second. We need a sampling rate of at most $1:100$, i.e., $p < 10^{-2}$. Assuming that the tested interval is at least 10 seconds long, more than six million packets pass through the switch during the interval, i.e., $N > 10^6$. From Equation 2 we get that the threshold, T, can then be roughly 10^{-3} or more.

Next we consider the fact that the flows that are monitored by exact counters are updated in batches (when reading the switch flow entry counters). To make sure that their counters in the approximate HH structure are not evicted between updates, we set the number of entries, v, to be high enough considering the threshold, t, for monitored flows.

Next we show that by choosing $v = 2/t$ the number of samples that would cause the eviction of one of the monitored flows, that is, a flow that is located at the top part of the approximate heavy hitters structure, is very high.

Assume we have k monitored flows, the sum of their counters is at least $k \cdot n \cdot t$. The number of other values in the table is $v - k$, and their sum is at most $n - knt$. In order for the minimal monitored flow to be evicted, all lower values in the table should exceed it, i.e., all smaller counts need to become higher than nt. Their sum should thus be at least $(v-k)nt$, increasing by at least $(v-k) \cdot nt - (n - knt) = vnt - n$. Since the counts change by the number of incoming samples, if we set $v = \frac{2}{t}$ then the number of new samples received between batch updates should be as large as the number of all samples received so far (n) which is highly unlikely.

3.2. *Interval Heavy Flow and Bulky Flow Detection*

Recall that, an *interval heavy flow* is a flow whose volume is more than T percent of the traffic seen in the last time interval of length m. While the problem is defined in a continuous manner, that is, an interval can begin at any point in time, considering the inherent subtle delays caused by the OpenFlow architecture, an approximate solution is sufficient.

Item	Counters				
	Accumulative	c_0	c_1	c_2	c_3
a	106	4	78	4	20
b	98	11	24	7	56
c	51	5	10	27	9

Figure 2: The modified heavy hitters data structure using counter arrays. In this example the active counter is currently c_1.

Our solution makes use of the Sample&Pick algorithm, specifically we take the array of counters in the heavy hitter module in the controller as the starting point. We modify this structure so that instead of maintaining one counter per item (flow), an array of counters is maintained for *each flow* that is kept in the heavy hitter module. In addition, for each flow we maintain an additional accumulative counter. The updated counter structure is depicted in Fig. 2.

The array of counters for each flow maintains the history of the flow's counter values in fixed intervals of time. The flow's accumulative counter is the sum of all the counters in the flow's array. Let m seconds be the selected time interval, and let there be r history counters maintained for each flow, we get a sub-interval that is $\frac{m}{r}$ seconds long. The basic idea is that in each sub-interval a different counter in the array is updated by the HH module, in addition to updating the accumulative counter. Thereby, consecutive (cyclicly) counters in the array can be used to calculate the number of times the value appeared in the entire interval. At the beginning of the sub-interval, for each flow, the value of the active counter is decreased from the value of the accumulative counter. Then all active counters in all flows are reset to zero. In this manner, at the end of each sub-interval, for any flow, the active counter equals the number of times the flow was sampled during that sub-interval, and the value of the accumulative counter equals the number of times the flow was sampled in the last interval m. It follows that if the index of the active counter is a s.t. $0 \leq a \leq r-1$ for any $r' \leq r-1$ the sum of the cyclically consecutive counters

between index $a - r' \mod r$ and a equals to the number of times the item was seen during the r' previous sub-intervals.

Note that if an interval does not begin at the beginning of an exact sub-interval, we will consider it to begin at the start of either the current or the consequent sub-interval.

The accumulative counter has two additional important uses: 1) it is used to maintain the threshold ratio; 2) it is used by the heavy hitters algorithm as the de-facto counter for deciding which flow has the minimum counter and should be evicted.

Using the accumulative counter in this manner is the basis for the correctness of our algorithm, which we will now briefly show. Given an interval i of length m, denote N to be the number of items seen in i. If i is made up only of whole sub-intervals, it is easy to see that at the end of interval i the accumulative counter of each flow in the structure is equal to what its counter would be had we reset all of the counters at the beginning of the interval. Therefore, using the accumulative counters as described above provides us with a heavy hitters mechanism which supports the same counter error rate (i.e. $\frac{N}{v}$) as that of [8]. If, however, i begins in the middle of a sub-interval, the counter error rate is slightly higher. In this case, suppose i contains j complete sub-intervals, and at most 2 partial sub-intervals. The additional error contains appearances of the flow which occurred in the partial sub-intervals, which may incur an additional error of at most $\frac{N}{v}$ since otherwise it would be heavy for an interval comprised of only complete sub-intervals as well, making the overall error rate in this case to be $\frac{2N}{v}$.

Notice that bulky flows can be detected by using the above mechanism without dividing the counters sum by the relevant sum of counters, but rather taking the absolute values.

4. Traffic Sampling

An SDN controller sets flow entries in the switch, a flow entry can match one or many flows but generates one statistical record for all matching flows. A controller has to install a flow entry per each separately monitored flow in real time by sending all unmonitored flows to the controller which in turn would install a specific entry for each. Monitoring flows in real time in the controller is infeasible due to controller computation speed constraints. Therefore in order to find large flows in SDN networks, sampling has to be used to reduce the set of monitored flows.

We discuss two types of traffic sampling: packet sampling and pseudo byte sampling, for which we provide the following definitions respectively:

Definition 4.1. ***Packet sampling:*** *Select each packet in a stream of packets with probability p,* $0 \leq p \leq 1$.

Note that the number of packets sampled from each flow times $1/p$ is an estimation of the real number of packets in the flow (during the sampling period).

Definition 4.2. ***Pseudo byte sampling:*** *Select each byte in a stream of traffic with probability p,* $0 \leq p \leq 1$.

Practically, this translates to a packet size based sampling, where given a stream of packets, a packet of size s bytes is selected with probability $1-(1-p)^s$. For small enough p, this can be approximated by $1-e^{-ps}$ and since usually $ps << 1$ it is approximated by simply $p \cdot s$. With this type of sampling, the number of packets sampled from each flow times $1/p$ is an estimation for the real number of **bytes** in each flow (during the sampling period).

4.1. Packet Sampling

We present two approaches for packet sampling, each using different SDN features.

Packet Sampling Using Random Selection: The following technique in the most direct way to implement packet sampling, it utilizes OpenFlow weighted groups (Section 7.3.4.2 in [16]), an optional feature intended for unequal load sharing and we expect it to be supported by future P4 compilers.

A weighted group contains a list of buckets each with different weight and actions. A packet is assigned to such a group (by the apply_group instruction) is randomly diverted to one of the buckets according to the weights and that bucket's actions are applied to the packet.

In our case, we use a group with two buckets - an "active" bucket that transfers to the receiver and a "dummy" bucket does nothing. We set the weights of the buckets according to the sampling probability p: weight 1 for the active bucket and weight $\lceil \frac{1}{p} \rceil - 1$ for the dummy bucket.

Note that as weighted groups are optional in the OpenFlow standard and are currently considered expensive (in terms of switch resources) and are not supported by P4, this solution is not compatible with all switches.

A similar sampling technique can be achieved by using OpenFlow round-robin groups, where for each packet the next action bucket is chosen (in round robin order). This technique is less compatible and more expensive than weighted groups based technique, and we therefore only use it for comparison with other techniques without describing the full implementation details.

Packet Sampling Using Hash Matching: As random generators are not natively supported by current SDN standards, OpenFlow and P4, we suggest to use the

hash of the packets instead. More precisely we suggest to use Ethernet CRC or TCP/UDP checksum fields and match them against predefined bit patterns thereby selecting which packets to sample and send to the collector. We overcome weaknesses of this method in the sequel.

More precisely, assuming $p = \frac{1}{2^k}$ the controller randomly selects a ternary pattern with k $0/1$-bits (not '*'s) for matching the checksum field, and install a flow entry with that pattern as the match and with an action to forward to the collector. For example, sampling with probability $p = 2^{-13}$ 0.0001 is implemented by matching the (16 bit) checksum to a ternary pattern with 3 $*'s$ (don't cares) and 13 zero/one bits.

To achieve uniform packet sampling using this mechanism, the selected pattern must be chosen with a uniform distribution and on a packet field which is uniformly distributed across all packets. The first requirement insures that any pattern value may be chosen with equal probability, and the second requirement insures that all packets have an equal chance of matching this pattern.

Matching unconventional packet fields (e.g. checksum) is supported in P4 and is also supported by some SDN switches such as the NoviKit [6, 7] using the optional Experimenter extension. In general this method uses the fundamental properties of all match-action modules (flow tables, TCAMs, etc..) and therefore expected to be easily realized in future network devices and control protocols.

Note that setting a single match pattern without changing it may present some problems. For example, crafted packets such as those in DDoS attacks may be missed. Such packets may be generated with a specific checksum value, and would be missed by this method. If, for example, this attack accounts for 50% of all traffic going through a link for some given time interval, and these packets all have the matched field set to some predetermined value, then the sampling is no longer done uniformly across all packets causing the sampling rate to be reduced in half. In order to deal with such scenarios, the controller should modify the selected match pattern randomly every fixed period of time, so that the mechanism approximates a sampling with a uniform probability for selecting any packet over a long enough period of time.

Additionally, since each change in the bit pattern requires a new rule (e.g., OpenFlow FlowMod command) to be sent by the controller to the switch, there is a tradeoff between the safety of the scheme and the control traffic it creates. It is also possible to send multiple commands in batches utilizing rule timeouts to set the end time of rule liveness, yet these rules have to be separated by additional rules to set the start time. Considering a short 1sec update interval and a command packet size of 108 bytes (40 bytes for TCP/IP headers and 68 bytes for OpenFlow 1.3 FlowMod message with two actions) we get an insignificant control plane traffic of 108B/s (in each direction).

The flow entries can be installed in a dedicated flow table, so that the sampling process does not interfere with other switch processing. Packets that match the pattern are sampled and then *all* packets continue to traverse the rest of the tables as in the unmodified pipeline. This process is depicted in Fig. 3.

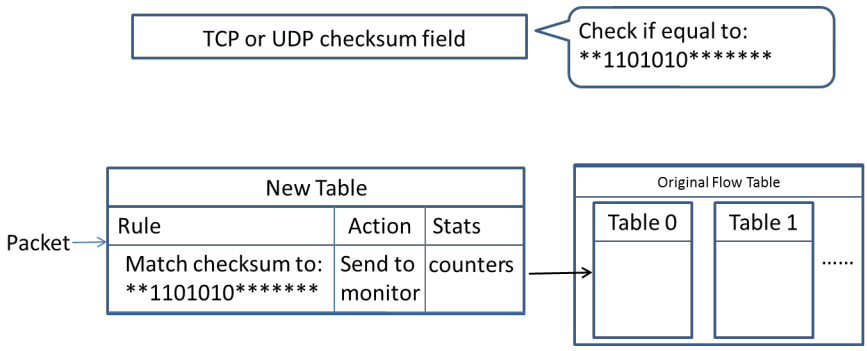


Figure 3: Example of the *randomized bit* algorithm for packet sampling. All packets traverse through both the new table and the original flow table. The sampling rate provided is $p = \frac{1}{128}$. Sampled packets may be sent to a monitor or the controller.

4.2. Pseudo Byte Sampling

As described above, *Pseudo Byte Sampling* with probability p per byte is approximated by sampling each packet with probability $p \cdot s$, where s is the packet size.

We present optimized techniques for pseudo-byte sampling, which are based on matching the packet size. Matching unconventional packet fields (e.g. packet size) is supported in P4 and is also supported by some SDN switches such as the NoviKit [6, 7] using the OpenFlow optional Experimenter extension

A General Approach for Pseudo-Byte Sampling: A straightforward implementation of the pseudo-byte sampling is to use multiple instances of any of the packet sampling implementations presented so far, where each instance samples with a different probability, and we divert each packet to the most accurate sampling instance considering the packet size. More formally, given a set of packet sizes $\{s_i\}_{1 \le i \le R}$, we define the set of sampling instances $\{PS_i\}_{1 \le i \le R}$, where PS_i samples any packet with probability $p \cdot s_i$. Moreover, we divert each packet with size s to the sampler PS_z, where $z = argmin_i |s - s_i|$.

The maximum error ratio in this method is $max_i \frac{s_i}{s_{i-1}}$. Therefore, in order to bound the error s_is should be chosen as geometric series. For example, for $1 \le i \le R$, $s_i = m \cdot 2^i$ where m and M are min and max packet size (e.g., 64 and 1500 for

Ethernet) and $R = \log_2 \frac{M}{m}$. Finally, following last example, given a packet of size s we divert it to the $PS_{\lceil \log_2 s \rceil}$.

Note that this approach presents a tradeoff between accuracy and resources. In order to reduce the maximum error, one has to use more sampler instances.

Pseudo-Byte Sampling with Hash Comparison: The following sampling technique uses constant resources and has optimal accuracy. It is fully supported by P4 and is also supported by some SDN switches such as the NoviKit [6, 7] using the OpenFlow optional Experimenter extension.

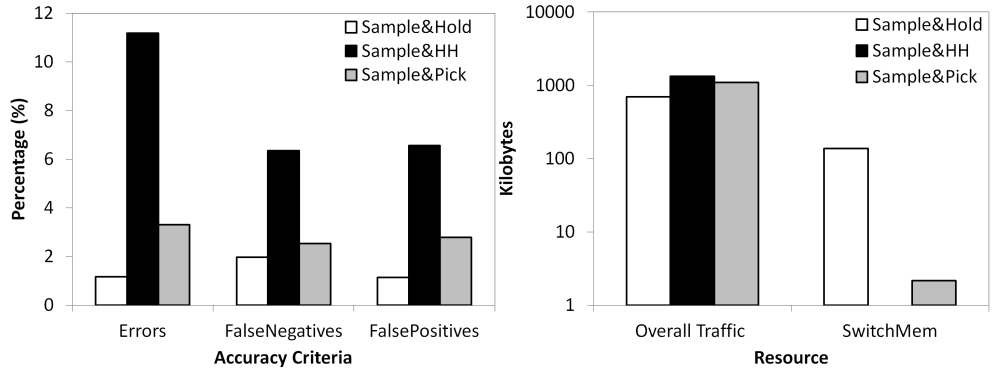
Before describing the technique we first make the following observation: if s and M are numbers such that $0 < s < M$ and x is a random variable chosen uniformly from $[0, M]$, then the probability that $x \leq s$ is s/M, i.e. for $x \sim U([0, M]), Pr(x \leq s) = \frac{s}{M}$. Following the last observation, if we substitute M with $\frac{1}{p}$, we get that the probability of sampling a packet of size s, namely ps, is equal to the probability that $x < s$. This means that given access to such uniform distribution we can implement size based sampling in the following way: for each packet of size s, first randomly choose x, then if $x < s$ transfer the packet to the receiver.

Similarly to the hash matching technique, we suggest to use the packet checksum as a random number generator. Assuming $\frac{1}{p} = 2^b$, where $b \in \mathbb{N}$, we use the first b bits of the checksum field as the random variable x, and we define rules that check whether $x < s$. If the comparison succeeds the action should forward the packets to the receiver and otherwise do nothing.

Comparing two fields is also not natively supported in OpenFlow but can be implemented by a flow table filled with $2b + 1$ rules, where b is the width of the compared numbers (in bits) [17].

Similarly to the packet sampling with hash matching described in the previous section, the pseudo-byte sampling technique presented here might miss specific classes of crafted packets whose checksum is high. Therefore we need to add some external randomness that is changed over time. While in the case of packet sampling we can just change the pattern, in the pseudo byte sampling case we need to affect the comparison result.

The solution we suggest is that every fixed time interval the controller will modify a rule (or a batch of rules with different timeouts) that writes the metadata field of every packet with some value r, and that value will be used in a modified version of comparison that checks whether $x \oplus r < s$. For each new value of r the controller needs to send one FlowMod command packet whose size in our scenario is less than 110 bytes. As in the packet sampling case, even for short time interval of 10 seconds, the control traffic overhead is insignificant.



(a) Comparison of algorithms by Counter error, False negative errors and False positive errors.

(b) Comparison of algorithms by Overall traffic (between switch and controller) and Switch memory usage.

Figure 4: Resource consumption and accuracy comparison

5. Evaluation

5.1. Comparison of Algorithms

5.1.1. Comparison of Our Algorithms

We compare our Sample&Pick algorithm to the two additional solutions described above Sample&Hold and Sample&HH (See algorithms overview in Table 1). We analyze the resource consumption and accuracy of each of the algorithms in fixed time intervals. We use 10 intervals of 5 seconds each, and we collect the counters of each algorithm at the end of each interval. In addition we compare the results of these algorithms to that of the OpenSketch Heavy Hitters detection mechanism [5]. For our analysis, we use a one-hour packet trace collected at a backbone link of a Tier-1 ISP in San Jose, CA, at 12pm on September 17, 2009 [18].

We chose the following simulation parameters $T = 5 \cdot 10^{-3}$ (to detect flows each of which takes up more than 0.5% of the traffic), $p = \frac{1}{1024 \cdot 10^2}$ Bytes (to sample roughly 0.1% of not picked packets) and considering the analysis at Section 3.1.4 we set Sample&Pick parameters $t = 2 \cdot 10^{-3}$ and $v = 2000$.

Figure 4a shows a comparison of the three algorithms based on accuracy criteria. The counter error refers to the ratio between the real count of the heavy hitters and the algorithm estimates. The false negative and false positive errors is the ratio between Heavy Hitter (HH) flows missed to the total number of HH flows, and the HH flows wrongly detected to the total number of HH flows respectively. Figure 4b shows a comparison of the three algorithms based on the amount of traffic

they generate and the amount of memory they use in the switch. As can be seen, while Sample&Hold provides the best accuracy results, it requires an increasing amount of counters and therefore its switch memory consumption is significantly higher than that of the other algorithms. In contrast, Sample&HH requires the least amount of switch memory since all of the heavy hitters computation is performed in the controller yet it relies on sampling alone and provides significantly lower accuracy results. Our testing shows that Sample&Pick provides accuracy results only slightly inferior to those of Sample&Hold yet requires significantly less switch memory.

Technique	Error Rate	Switch memory usage	Controller $\leftrightarrow$ Switch Traffic
Sample&Pick	3.3%	$2KB$	$220KB/s$
Sample&Hold	1.15%	$400KB$	$140KB/s$
Sample&HH	11.3%	$\leq 1KB$	$270KB/s$

Table 3: Resource consumption test results

As can be seen in Table 3, Sample&Hold gives the smallest error rate, since it performs an actual count of all flows that it samples, yet it uses significantly more switch memory. Sample&HH uses only samples for the counter estimates without using any counters in the switch yet incurs significantly higher error rates. Sample&Pick has relatively small error rates due to the actual counting of potentially heavy flows, yet due to the careful selection of which counters to place in the switch, the switch memory usage in Sample&Pick is very low. According to our testing, the error rate of Sample&Pick may be further reduced with increased sampling rate or counter polling rate, yet the switch memory requirement remains steady at $2KB$ as determined by our parameters. The controller$\leftrightarrow$switch traffic (sum of traffic in both directions) of each of the presented algorithms is directly influenced by the sampling rate (recall in this case $p = \frac{1}{1024 \cdot 10^2}$ Bytes) and the counter polling rate of the controller. In the case of Sample&Pick the polling rate is set to be every 0.1 seconds in these tests, while in Sample&Hold the controller only polls for the counters once at the end of the interval. As can be seen, Sample&HH produces a larger traffic overhead since all sampled messages are sent to the controller whereas in the other two algorithms the counters in the switch perform the aggregation locally.

5.1.2. Comparison to Existing Solutions

We compare our results to several existing solutions: Sampled Netflow [2], OpenSketch [5], Hashpipe [19] and UnivMon [20].

We compare the results achieved by the different solutions when tested on ISP backbone traffic. We analyze the results presented by the authors in each of the above papers using multiple different CAIDA traces from recent years.

Summarizing our testing of our Sample&Pick algorithm, it achieves an average error rate of 3.3% using $2KB$ of switch memory. Both false positive and false negative rates are approximately 2% and the communication overhead is averaged at $220KB/s$.

We now compare these findings to those of the other solutions. Sampled Netflow [2] is based on sampling performed at the switch and processing of the data at the collector. Our Sample&HH algorithm is a simulation of this mechanism. Sampling based methods reduce communication overhead at the cost of accuracy[15]. As can be seen in Table 3, the Sample&HH mechanism achieves a significantly higher error rate of 11.3%, and still requires more communication between the switch and the controller. False positive and False negative are both higher at $\geq 6\%$.

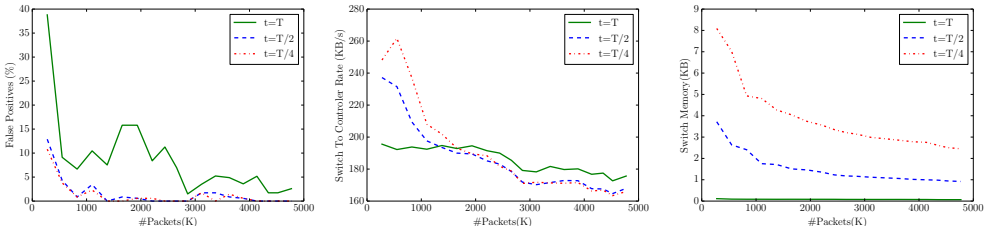
Comparing our results to testing done on the OpenSketch Heavy Hitters detection mechanism [5], we can see that to achieve a comparable error rate of approximately 3% Opensketch utilizes $\geq 100KB$ of memory in the switch compared with $2KB$ needed by our method. To achieve better error rates, OpenSketch requires hundreds of KB more. The false positve and false negative rates are $\leq 1\%$. The table maintained by OpenSketch at the switch needs to be periodically sent to the controller therefore incurring a communication overhead proportional to the amount of memory needed in the switch or greater.

UnivMon [20], a *P*4 based solution, achieves slightly better error rates of $\leq 1\%$, yet the switch memory requirement is averaged at $\geq 200KB$ which is significantly higher than that required by our solution. The communication overhead includes sending the sketch to the controller every interval.

Haspipe [19], is another *P*4 based solution, performed completely in the data plane. It reaches an estimation error of $\leq 10\%$ and false negative rate of $\leq 10\%$ using $80KB$ of switch memory. Communication overhead includes reporting the detected heavy hitters to the controller.

5.2. Parameters Evaluation

We evaluate the affects of different parameters on our system. In our analysis, we compute average results for 50 time intervals, 10 seconds each, of packet traces collected at backbone links of a Tier-1 ISP in San Jose, CA and Chicago, IL, during the years 2009-2016 [18, 21, 22, 23].



(a) Comparison of error (false positives) rate and convergence.

(b) Comparison of control traffic rate from switch to controller.

(c) Comparison of switch memory (monitoring rules).

Figure 5: Affect of varying t values

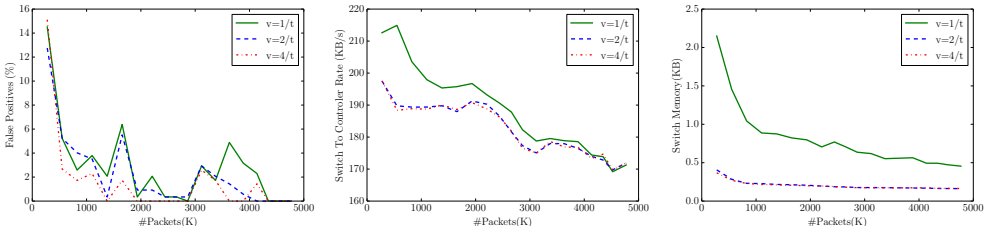
The base setting of our system for all the following tests used the following parameters: $T = 0.01$, $p = \frac{1}{256}$ Packets, $t = 0.005$, $v = 400$.

The first parameter we examine is t, which is the threshold for considering a flow a potentially large flow and installing an exact monitoring rule for a it. As discussed in Section 3.1.4 we recommend using $t = T/2$, where T is the target threshold for large flow. In Figure 5a we can see that when setting t as high as T we have more errors, this is due to the fact that exact counters are placed for bigger flows and exactly at target, therefore less flows are accurately monitored, and more flows might be measured by random sampling, hence decreasing accuracy. However as seen in Figures 5b and 5c, this setting requires less communication with the controller at the beginning (as less counter values are sent) and less monitoring rules at the switch.

When setting t to be as low as $T/4$ we can see in Figure 5c that it results in high switch memory consumption, as more monitoring rules are installed, without noticeable improvement in accuracy compared to setting $t = T/2$ (Figure 5a). Overall, we can conclude that choosing $t = T/2$ provides a good balance of monitoring resources and accuracy.

Next, we consider different values of parameter v (the number of items maintained by the heavy hitter module in the controller) and its effect on the system. As discussed in Section 3.1.4 we recommend setting v sufficiently higher than $1/t$, for example $v = 2/t$, as $1/t$ is the maximum number of monitored flows in the switch.

As can be seen in Figure 6a, using lower values of v may decrease the accuracy during the inspected 10 sec window but not in the final result. However, as can be seen in Figures 6b and 6c, using $v = 1/t$ produces significantly more communication to the controller and consumes more switch memory compared to $2/t$ and $4/t$. We can conclude that using $v = 2/t$ produces good results with low controller memory consumption.



(a) Comparison of error (false positives) rate and convergence.

(b) Comparison of control traffic rate from switch to controller.

(c) Comparison of switch memory (monitoring rules).

Figure 6: The effect of varying the value of v.

Note that while v represents the memory overhead of the controller, the processing time overhead of the controller is proportional to the amount of incoming traffic to the controller. This corresponds to the number of counters and packetIns sent from the switch, each requiring an update to the table of counters in the controller.

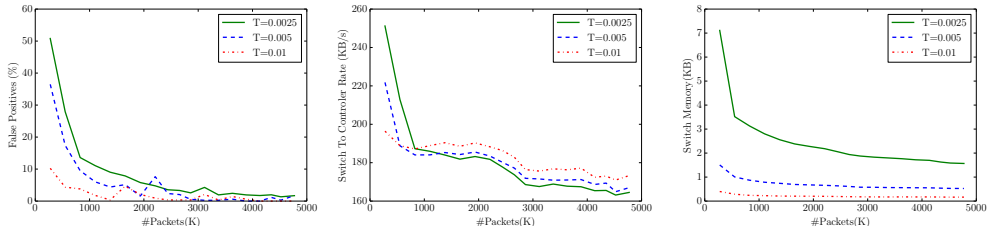
In addition, note that other factors were also measured during the executions, such as errors in the reported large flow counts and traffic from the controller to the switch, however due to space constrains, we included only those that were most affected by the studied parameters. Expected values for the missing measures can be seen in Table 3 and Figure 4a.

Finally, we compare the performance of our scheme for different values of the large flow threshold T, while using the suggested values for v and t. As can be seen in Figure 7a, with smaller values of T it takes a longer time to achieve accurate results (although all runs complete with a very small error rate). This is due to the fact that a lower value of T calls for the detection of smaller flows which are more prone to errors caused by sampling.

Considering the traffic from the switch to controller, as can be seen in Figure 7c, the smaller the value of T, the less PacketIn messages generated. This is due to the fact that a smaller T means more monitored flows and therefore less traffic is sampled and sent to the controller. However, as seen in Figure 7c, more monitoring rules means higher consumption of switch memory.

6. Distributed Setting

In many cases, in order to achieve a comprehensive view of the network, it is required to distributively monitor traffic at multiple switches. There are two main challenges to deal with when detecting large flows in this distributed setting; false



(a) Comparison of error (false positives) rate and convergence.

(b) Comparison of control traffic rate from switch to controller.

(c) Comparison of switch memory (monitoring rules).

Figure 7: Results for different values of large flow threshold (T)

negatives due to split flows and false positives due to sequential flows. Split flows are large flows that their traffic is split to small sub flows, each going through a different monitoring switch, and therefore monitored in parallel. Split flows could be caused by load balancing between different paths, rerouting or flows that do not share the same source and destination IPs and are therefore routed through different paths in the network. On the other hand, sequential flows are small flows that each of their packets traverse multiple monitoring switches and are therefore over sampled or counted.

In this section we extend our Sample&Pick solution in order to support this distributed setting. We describe the changes that need to be done to the sampling and to the large flow detection scheme. We note that our solution easily scales with the number of monitoring switches. To support multiple controllers, a hierarchy of controllers needs to be defined and data should be collected by the controllers and forwarded up the hierarchy.

Sampling: In order to handle over sampling of sequential flows, flows that each of their packets traverse multiple switches, we need to prevent each packet from being sampled more than once. We suggest to do so by marking packets after they are sampled (whether selected or not) and by applying sampling only to unmarked packets. Marking of packets can easily be managed in SDNs (with OpenFlow and especially with P4), for example by utilizing one bit in the VLAN tag. Matching the VLAN tag of each packet can be easily done and allows to skip sampled packets. Note that the marks should be removed at egress ports so that they do not affect the traffic leaving the network.

Heavy Flow Detection: As described in Section 3.1, our Sample&Pick algorithm makes use of both sampling and exact counter rules in the switch. To support the distributed setting, and to handle split flows, that each of their sub flows goes through a different monitoring switch, all of the samples and counter values from

all monitoring switches should be aggregated centrally by the controller. The controller will receive the samples and counter values from the different switches and treat them as if they were generated by a single monitoring switch. One of the implications of that is that when a flow becomes suspect of being large, exact counter rules should be installed on all monitoring switches, to assure that all consequent packets going through the network are counted.

Similarly to sampling, in case of sequential flows that traverse multiple switches, exact counters (on different switches) should not count the same packet more than once. The same packet marking technique we suggest to avoid over sampling, can be used in order to prevent multiple counting (see Figure 8), i.e., marked packets are not matched against exact counter rules nor sampled. Moreover, packets which match exact count rules are marked even if they have not been sampled.

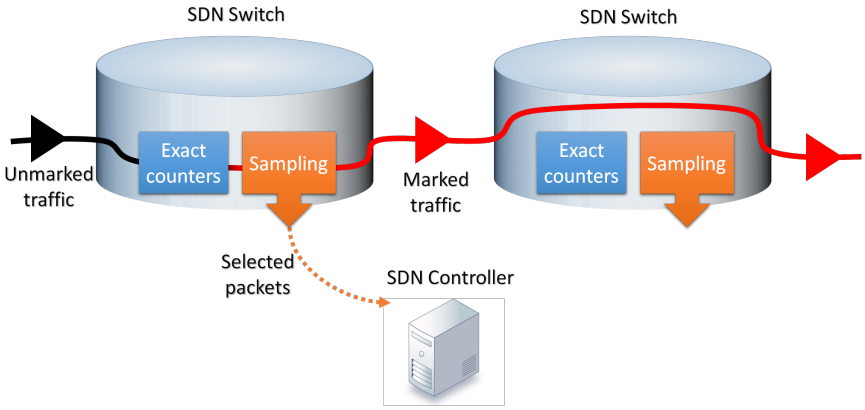


Figure 8: Marking sampled packets in the distributed setting.

7. Related Work

One of the earliest network monitoring tools was Cisco Netflow [2], which allowed collection of IP flow level statistics. Netflow provided the ability to gather information from the router about every IP flow, including byte and packet counts yet suffered from high processing and collection overheads, which were partially decreased using sampling in the variant *Sampled* Netflow, yet this variant provided reduced accuracy caused by the straightforward use of sampling [15]. In [15] Estan and Varghese significantly improve the accuracy of the sampling process by introducing the *Sample and Hold* algorithm which provides better accuracy while reducing the processing and collection overhead. The sample and hold algorithm

is essentially sampling with a "twist". As in regular sampling, each packet is sampled with some probability, and if there is no entry for the packet's flow, an entry is created. Once an entry for a flow exists, it is updated for every packet thereafter in that flow.

In a usual setup, monitoring devices are placed in central locations in the network (such as Arbor's Peekflow [24], or other security detection devices) and samples of traffic are being sent to the monitoring devices for various additional processing for which the switch/router are not suitable, such as heavy-hitters analysis, DPI, and behavioral analysis. These monitoring devices usually cannot absorb and process all the traffic. Therefore, traffic must be sampled, and only the samples or relevant flows should be forwarded to these devices.

As the networks evolved, network monitoring tools with more advanced capabilities were developed. In [25], for example, a flow monitoring tool was presented. There, they discussed adding flow sampling abilities as an inherent capability of the routers. They provide a framework for distributing the monitoring across routers, allowing for network-wide monitoring. By using uniform hash functions, flow sampling is not duplicated across different routers which route the same flow.

In OpenFlow the flow table allows us to define rules which support counting of bytes and packets per flow. However, this is not sufficient for more advanced measurements. Recently there have been several works that discuss or suggest enhancements to network measurement capabilities for both OpenFlow and for SDN in general. FleXam is a sampling infrastructure for OpenFlow proposed in [26], which adds sampling capabilities, using random number generation. Opensketch [5] is a commonly known measurement platform which provides a simple approach to collect and use measurement data, separating the measurement data plane from the control plane. The paper suggests a new architecture, where in the data plane, a pipeline of three essential building blocks is provided: hashing, filtering and counting, and in the control plane, a wide library of measurement tasks is provided. The above works suggest an alternate to the OpenFlow architecture, while our work relies on features that already appear in the current OpenFlow standard as required or optional features, in addition to the common extensions such as matching on an extra field in the packet. These extensions follow the concepts described in [27], that suggests that the OpenFlow standard should allow the user to configure the headers that the switch can examine. All our modification are in the spirit of OpenFlow architecture. A broader comparison to our solution is given in Section 5.1.2.

We note that there are works that do not require changes to the OpenFlow standard. For instance, OpenNetMon described in [28] is a controller module for monitoring flow level metrics, such as packet loss, delays and throughput in OpenFlow networks, for determining whether QoS criteria are met, which is based on

the OpenFlow standard. Our solution which combines both a switch module and a controller module provides accurate results while significantly reducing the communication overhead. Another such solution has been introduced in [29]. There, a method for heavy hitters detection is presented which analyzes network statistics using different aggregation levels. This solution does not require sampling which allows them to reduce communication overhead.

A recent work [30], proposes a method for distributing the monitoring tasks between different switches in order to reduce the number of rules needed in each switch. This method is orthogonal to our distributed solution (see Section 6), and can be combined to further reduce the number of switch entries.

An additional line of work relies on different hardware for heavy hitter detection. A recent work, [31], proposes DREAM, a framework for identifying heavy hitters (see Section 2.1) in traffic using TCAM based hardware. As shown in [31], the algorithm they use for heavy hitters detection may require more TCAM entries than a commodity switch may have available. Therefore DREAM performs efficient multi-switch resource allocation between switches to achieve the desired accuracy rates. The Sample&Pick algorithm we propose (Section 3.1) requires significantly less counters in the switch and can be used by DREAM to reduce the overall number of switch entries used.

Solutions have also been presented using a NetFPGA Openflow switch. In [32] a solution is presented which is based on the Count-Min sketch [11] with an auxiliary data structure for maintaining the identities of the heavy hitters. The memory required by this solution is slightly higher, yet comparable to the requirements of our proposed algorithm, however, it is not limited to the match-action model and assumes that the required data structures can be placed on a particular hardware design.

Recent advances in programmable data planes including the development of the P4 language and switch architecture have brought the development of telemetry solutions performed mainly or solely in the data plane. The Hashpipe algorithm presented in [19] provides a solution for heavy hitters detection done completey in the data plane using P4 programmable switches. A comparison to our solution is given in Section 5.1.2.

A generalized approach is presented in UnivMon [20] where a solution for a P4 universal sketch is presented which is performed mostly in the data plane. The solution maintains a sketch in the data plane and a summary of the sketch is sent as required for further evaluation in the control plane. A comparison to our solution is given in Section 5.1.2.

We note that there have also been works done on variations of the interval heavy hitters problem, such as [33] which proposes a scheme based completely on statically allocated memory for finding sliding window heavy hitters.

8. Conclusions

We have presented techniques for performing large flow detection and sampling in SDN. Our sampling techniques are unique in that they are simple and remain mostly within the confinements of the OpenFlow standard. Our approximation algorithms for large flows detection provide a generic mechanism for SDN, providing a way to detect various types of large flows with a relatively small error rate while minimizing the computation and space overhead in the switch and requiring little controller-switch communication. Furthermore, we expanded our algorithms to a distributed multi-switch setting.

9. Acknowledgments

We thank the anonymous reviewers for their insightful comments.

10. References

[1] "Kaspersky ddos intelligence report for q1 2016," https://securelist.com/analysis/quarterly-malware-reports/74550/kaspersky-ddos-intelligence-report-for-q1-2016/.

[2] [Online]. Available: http://www.cisco.com/c/en/us/tech/quality-of-service-qos/netflow/index.html

[3] Q. Zhao, Z. Ge, J. Wang, and J. J. Xu, "Robust traffic matrix estimation with imperfect information: making use of multiple data sources," in *Modeling of Computer Systems, SIGMETRICS/Performance 2006*.

[4] B. Stephens, A. L. Cox, W. Felter, C. Dixon, and J. B. Carter, "PAST: scalable ethernet for data centers," in *Conference on emerging Networking Experiments and Technologies, CoNEXT '12*.

[5] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX NSDI*, 2013, pp. 29–42.

[6] "NoviFlow's NoviKit," http://noviflow.com/products/novikit/(accessed on March 2015).

[7] "NoviFlow's NoviWare," http://noviflow.com/products/noviware/(accessed on January 2017).

[8] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *ICDT*, 2005, pp. 398–412.

[9] J. Misra and D. Gries, "Finding repeated elements," *Sci. Comput. Program.*, vol. 2, no. 2, pp. 143–152, 1982.

[10] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, 1999.

[11] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *LATIN*, 2004, pp. 29–38.

[12] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," *PVLDB*, vol. 5, no. 12, p. 1699, 2012.

[13] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *PVLDB*, vol. 1, no. 2, pp. 1530–1541, 2008.

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

[15] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[16] Open Networking Foundation, *OpenFlow Switch Specification Version 1.3.2*, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf

[17] Y. Afek, A. Bremler-Barr, and L. Schiff, "Orange: Multi field openflow based range classifier," in *ANCS*, 2015.

[18] "The caida ucsd anonymized internet traces 2009 - sep. 17 2009," http://www.caida.org/data/passive/passive_2009_dataset.xml.

[19] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *SOSR*, 2017, pp. 164–176.

[20] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *SIGCOMM*, 2016, pp. 101–114.

[21] "The caida ucsd anonymized internet traces 2012," http://www.caida.org/data/passive/passive_2012_dataset.xml.

[22] "The caida ucsd anonymized internet traces 2014 - mar. 20 2014," http://www.caida.org/data/passive/passive_2014_dataset.xml.

[23] "The caida ucsd anonymized internet traces 2012," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[24] A. N. Inc., "Peekflow," Aug. 2004, http://www.arbornetworks.com/products/peakflow.

[25] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "csamp: A system for network-wide flow monitoring," in *USENIX NSDI*, 2008, pp. 233–246.

[26] S. Shirali-Shahreza and Y. Ganjali, "Flexam: flexible sampling extension for monitoring and security applications in openflow," in *HotSDN*, 2013, pp. 167–168.

[27] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *SIGCOMM*, 2011, pp. 254–265.

[28] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *NOMS*. IEEE, 2014, pp. 1–8.

[29] L. Yang, B. Ng, and W. K. G. Seah, "Heavy hitter detection and identification in software defined networking," in *ICCCN*, 2016, pp. 1–10.

[30] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *HotSDN*, 2014, pp. 85–90.

[31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: dynamic resource allocation for software-defined measurement," in *SIGCOMM*, 2014, pp. 419–430.

[32] T. Wellem, Y. Lai, C. Cheng, Y. Liao, L. Chen, and C. Huang, "Implementing a heavy hitter detection on the netfpga openflow switch," in *LANMAN*, 2017, pp. 1–2.

[33] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *INFOCOM*, 2016, pp. 1–9.