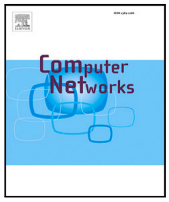
Computer Networks 196 (2021) 108239

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



FlowFight: High performance–low memory top-k spreader detection

Valerio Bruschi [a,*], Salvatore Pontarelli [b], Jerome Tollet [c], Dave Barach [c], Giuseppe Bianchi [a]

[a] CNIT/University of Rome Tor Vergata, Italy
[b] University of Rome La Sapienza, Italy
[c] Cisco Systems, United States of America

ARTICLE INFO

Keywords:
Superspreaders detection
Network monitoring
Software router

ABSTRACT

A recurring task in security monitoring/anomaly detection applications consists in finding the so-called top "spreaders" ("scanners"), for instance hosts which connect to a large number of distinct destinations or hit different ports. Estimating the top k scanners, and their cardinality, using the least amount of memory meanwhile running at multi-Gbps speed, is a non trivial task, as it requires to "remember" the destinations or ports already contacted in the past by each specific host. This paper proposes and assesses an innovative design, called FlowFight. As the name implies, our approach revolves on the idea of deploying a relatively small number of per-flow HyperLogLog approximate counters — only slightly superior to the target k — and involve the potentially huge number of concurrent flows in a sort of dynamic randomized "competition" for entering such set. The algorithm has been tested and integrated in a full-fledged software router such as Vector Packet Processor. Using either synthetic or real traffic traces, we show that FlowFight is able to estimate the top-k cardinality flows with an accuracy of more than 95%, while retaining a processing throughput of around 8 Mpps on a single core. We further show that FlowFight achieves same accuracy of the state of the art competitor SpreadSketch using 10x times less memory with 1.2x times higher throughput.

1. Introduction

Nowadays, performing line rate traffic monitoring is a very challenging task, since it requires significant computing and memory resources that prevent to easily integrate fast, scalable and flexible monitoring primitives maintaining high performance. In this paper, we address the problem of per-flow distinct counting, while maintaining high processing throughput and with a limited memory footprint. The goal is to automatically identify traffic anomalies that correspond to the detection of top "spreaders" ("scanners") [1] as the flows that exhibit the largest number of distinct connections. As an example, *spreaders* can be defined as hosts (i.e. source ip addresses) that contact a large number of distinct destinations (i.e. distinct destination ip addresses), as in the case of superspreader attacks [2,3]. This kind of task is of great interest, since flows with largest distinct items can bring about evidence of network anomalies such as port scanning, Distributed Denial of Service (DDOS) [4], or can be used for traffic engineering when specific routing policies are applied to superspreaders [5].

The problem of very efficiently determining the "spreader"/"scanner" nature of a target flow is closely related to the "cardinality estimation" problem pioneered by Flajolet and Martin in 1985 [6], and then further addressed and improved in many subsequent work, including [7,8]. Given a data stream which contains repeated elements, the goal of cardinality estimation (a.k.a. distinct counting) consists in finding *how many* of them are distinct. Notice that such problem differs from the common and popular top heavy-hitters detection [9], where flows are discriminated according to their size. Size estimation simply requires to remember *how many* elements in total are seen, whereas cardinality estimation requires to further remember, in some suitable and efficient way, *which* elements have been already seen to avoid double counting.

Our problem is even more challenging in network scenarios, as the spreading/scanning flows are *not* known before hands. Therefore we need to jointly (i) determine the top-k spreaders and (ii) estimate their cardinality, meanwhile using a limited amount of memory and providing high speed operation. However, the estimation of a single flow already requires hundreds of bytes of memory, even when advanced data structures such as HyperLogLog++ approximate counters are used [10], and if we had to deploy a single sketch for *each* possible flow, considering that the number of concurrently monitored flows in a high speed backbone may be in the order of millions, the memory required would be overwhelming.

To solve the scalability issue of the per-flow estimation, we developed FlowFight, an innovative algorithm that exploits approximate cardinality estimation algorithms based on HyperLogLog (HLL) sketch [8].

* Corresponding author.
E-mail address: Valerio.bruschi@uniroma2.it (V. Bruschi).

https://doi.org/10.1016/j.comnet.2021.108239
Received 1 December 2020; Received in revised form 19 May 2021; Accepted 8 June 2021
Available online 19 June 2021

As the name somewhat implies, to determine the top-k spreaders, FlowFight deploys a number of HLL sketches only slightly larger than k, and — loosely speaking — it engages flows in a sort of randomized "battle" for entering the top-k range. To this purpose, we designed a computational/memory efficient data structure and an original randomized access policy that can integrate the HLL structure for cardinality estimation on multiple flows. In this way, FlowFight is able to detect top spreaders and at the same time report the estimation of their cardinalities, while maintaining high performance and low memory. Thus, the research contributions of this paper are:

- We designed a memory efficient data structure based on a modified version of Stream Summary [11] for top-k spreader detection.
- We developed a specific access policy to identify which flows are good candidates to be inserted in the data structure.
- We defined a fast and memory limited rough estimation method to quantify, sort and select flows on the fly inside the data structure.
- We developed FlowFight inside the VPP framework [12] to validate performance in terms of packets processed per second, top-k precision and HLL relative error. This allows to test our solution in a full-fledged production grade software router.
- We compared FlowFight against a state of the art competitor sketch, called SpreadSketch [13], showing FlowFight able to achieve same accuracy of SpreadSketch with 10x times less memory and 1.2x higher throughput.

The rest of the paper is organized as follow. Section 2 presents the state of the art and discusses the issues related to top-k spreader detection, while Section 3 illustrates the proposed algorithm. In Section 4, we assess the algorithm comparing it with the ideal situation of using unlimited memory. And we report several micro-bench simulations used to tune the algorithm parameters. After, Section 5 reports experimental results in which the FlowFight algorithm is tested with both synthetic and real traffic. Finally, discussion about future work on FlowFight are provided in Section 6, while conclusions are drawn in Section 7.

2. Background, state of the art and challenges

For self-containment, we report the state of the art for both the cases of single-flow cardinality estimation, top-k heavy hitters detection and top-k spreaders detection. In this way, we can further remark the difference between measurement tasks, as well as illustrate the lessons learned and how to exploit them in our scenario.

2.1. Cardinality estimation (distinct counting)

Problem statement. Formally, we can define the cardinality estimation problem as follows: let $E = \{e_1, \ldots, e_N\}$ be a multi-set of N items where each item e_i is a member of a universe of U possible values and multiple items may have the same value. Let $m_i = \|\{j : e_j = i\}\|$ be the number of occurrences of value $i \in U$ inside the multi-set E. The number of distinct values in E is the number of elements from universe U appearing at least once in E (i.e. $m_i \neq 0$).

In short, cardinality estimation needs to remove duplicates, which makes it a more difficult problem since it has to somehow "remember" the observed elements for duplicate removal, while measuring a flow size only needs a counter (i.e. m_i).

State of the art. The naïve way to perform distinct counting consists in storing all the processed elements in memory, consequently it requires a memory footprint linear to the cardinality of the dataset. Obviously, the memory needed is too much and this approach does not scale. Consequently, many algorithms have started using an approximate approach to reduce resource/memory consumption. This results in an accuracy that is not exact but with a relative error to minimize. In literature, the typical approaches can be classified by two broad categories: sample based and sketch based. While the former does not scan the entire traffic dataset but just a subset, the latter scans the entire traffic and hashes each element into a sketch. Although it may seem that sampling can scale better, it can result in reducing accuracy on small time scales. While, sketches can be accurate even in very small time scale by employing in parallel independent sketches and hash functions to reduce the estimation error. Then, the sketch is usually used at query time to estimate the distinct count. Typical sketch-based algorithms[1] include PCSA [6], Multiresolution Bitmap [16] (a generalization of Linear Counting [17]), MinCount [18,19], LogLog [7], HyperLogLog [8] (and its derivatives [10,20]).

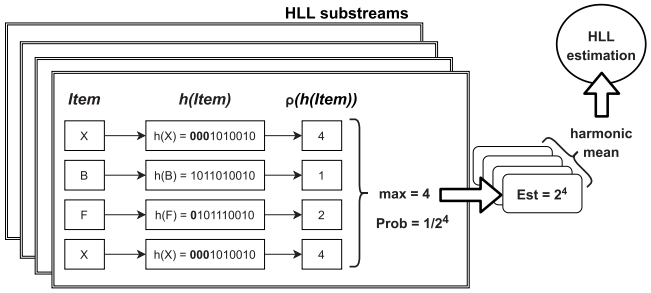


Fig. 1. Scheme of the HyperLogLog (HLL). Each substream performs the probabilistic counting method. Then, the harmonic mean is performed across all the substream estimations.

In this paper, we will focus on the HyperLogLog algorithm that has a very small memory footprint and an excellent computing performance, being suited for both hardware and software implementations.

HyperLogLog (HLL). HLL is an efficient structure to estimate the cardinality of a set and it is sketched in Fig. 1. It is based on the use of hash functions and requires a very limited amount of memory. HLL is based on the probabilistic counting method developed by Flajolet and Martin [6]. To understand the HLL, we start from the description of the probabilistic counting method. The counting method performs the hash $h(x)$ of an incoming item x and estimates the number of distinct items depending on the value of $h(x)$. In particular, we compute $\rho(h(x))$, where the function ρ returns the position of the leftmost 1 in the binary representation of $h(x)$[2]. After, it stores the maximum between the current max value and the new $\rho(h(x))$ value. At the end of the computation, the number of distinct items can be estimated as $2^{max_i(\rho(h(x_i)))}$. In fact, the probability that the function $\rho(h(x))$ gets a specific value n is 2^{-n}, since each bit in the hash has been generated uniformly. Hence, from a statistical point of view, after seeing n distinct elements, the function $max_i(\rho(h(x_i)))$ roughly approximates the log_2 of the number of distinct elements. However, the simple method explained above suffers of a large error variability due to the use of a single memory element where the maximum ρ value is stored. The HLL supersedes this limitation by: (i) dividing the input stream in m substreams and associating a register to each substream, and (ii) performing the harmonic average among the results collected by the different m substreams. Authors in [8] have shown that the HLL can provide a relative accuracy (the standard error) in the order of $1.04/\sqrt{m}$. As an example, HLL can estimate cardinalities of 10^9 with a typical accuracy of 5% using a memory of only 256 bytes. The small footprint and the use of hardware friendly primitives like hash functions suggest that the HLL can be easily used as a technique to estimate cardinalities without compromising the performance.

[1] Interested readers can have a look to these surveys [14,15] to investigate specific difference between several algorithm categories.

[2] We followed the original definition of $\rho()$ used in [7], where position start from 1.

2.2. Top-k heavy-hitters detection

Problem statement. Formally, we can define the top-k heavy-hitters detection problem as follows: let $P = \{p_1, \dots, p_N\}$ be a network stream with N packets. Each packet p_j belongs to a flow, where $f_i \in F = \{f_1, \dots, f_K\}$ and F is the set of flows. Let n_i be the number of packets belonging to the flow f_i in F. We order all flows $\{f_1, \dots, f_K\}$ so that $n_1 \leq n_2 \leq n_K$. Given an integer k and a network stream P, the output of top-k is a list of k flows from F with the largest flow sizes, i.e., $\{f_1, \dots, f_k\}$.

State of the art. Traditional solutions to find the top-k follow two basic strategies: *count-all* (i.e. reling on a sketch) and *admit-all-count-some* (i.e. just counting some flows).

The *count-all* strategy uses sketches (such as the CM sketch [21] or the Count sketch [22]) to record the sizes of all flows, and uses a min-heap to keep track of the top-k flows, including the flow IDs and their flow sizes. These methods are typically optimized for hardware implementation as they use statistically allocated memory and do not store explicit flow identifiers.

The strategy of *admit-all-count-some* (e.g. Lossy counting [23], Space-Saving [11], and Compact-Space-Saving [24]) is to admit all new flows while expelling the smallest existing ones from the summary. Therefore, this strategy requires the definition of an access policy that selects under which conditions a new flow must replace the smallest one among the k monitored ones. And to give new flows a chance to stay in the summary, their initial flow sizes are set as $n_{min} + 1$. Such a strategy drastically over-estimates sizes of flows, and this approach is unfeasible with sketches, since the distribution on the different sketch-register depends on the statistical properties of the flow monitored. However, an interesting representative algorithm of this category is Space-Saving which designs a data structure called Stream-Summary able to incur $O(1)$ overhead to search or update a flow, or find the smallest flow. Finally, this algorithm has further been re-designed to require considerably less space by using array indices instead of memory pointers: called Compact Space-Saving (CSS) [24].

2.3. Top-k spreaders detection

Problem statement. Now we can formally define our problem as follows: let $P = \{p_1, \dots, p_N\}$ be a network stream with N packets. Each packet p_j belongs to a flow, where $f_i \in F = \{f_1, \dots, f_K\}$ and F is the set of flows. Furthermore, each packet contains an item $e_w \in E$. We define the cardinality of f_i ($C(f_i)$) as the number of distinct items e_w contained in the packets in the flow i. We order all flows $\{f_1, \dots, f_K\}$ so that $C(f_1) \leq C(f_2) \leq C(f_K)$. Given an integer k and a network stream P, the output of top-k is a list of k flows from F with the largest flow cardinalities, i.e., $\{f_1, \dots, f_k\}$.

Discussion. In our scenario, we would like to take advantage from both the strategies deployed in top-k heavy-hitters detection: since from one side, a sketch is needed to estimate the cardinality but state of the art sketches cannot distinguish between flows, and on the other side we do not want to count every single flow.

To design a top-k spreader detection, we also need to take into account the effect of the approximation introduced by the per-flow estimation method. In fact, small cardinality difference between flows can significantly affect the top-k detection and make more difficult to achieve good accuracy. As an example, in Fig. 2, we report the cardinality for the top 200 flows of a real trace from CAIDA dataset. In the zoomed area, what happens around the 100th flow is shown. The colored area represents the 5% error to respect the exact value that can be introduced by the HLL estimator. It is worth to notice that in the case of detecting the first 100 flows, due to the estimation error, flows between the 93rd and the 107th position can be detect in the top-k. In Section 3, we will describe how we leverage the lesson learned by strategies described above to build an efficient top-k spreader detection algorithm, called FlowFight.

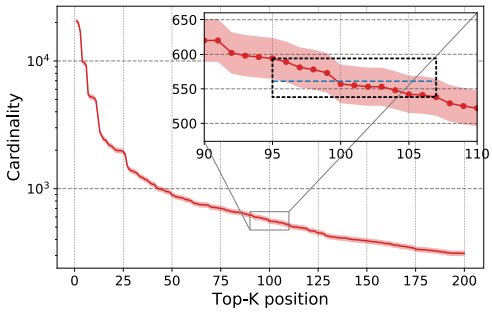


Fig. 2. Cardinality use case analysis on a real CAIDA trace. The colored area represents a 5% error for a typical estimation. In the case of 100 top-k flows, we report the border area in which due to the estimation error, the flows between the 95th–107th index can be interchangeable each other. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

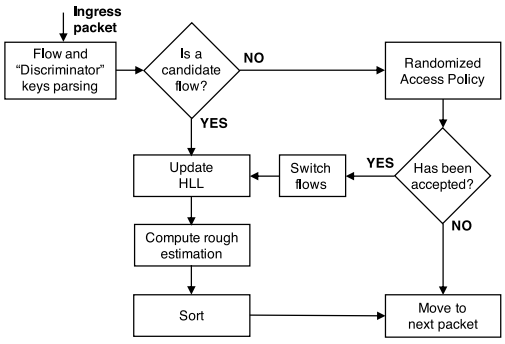


Fig. 3. The FlowFight scheme when it processes ingress packets.

Related work. Different strategies to detect top-k spreaders have been proposed in literature, such as [2,3,13,25]. Authors of vHLL [25] propose to allocate one estimator per flow in a virtual way, where the actual registers are shared among several virtual estimators. The vHLL data structure can use a compact memory space (down to 0.1 bit per flow on average) to estimate the cardinalities of flows with wide range and reasonable accuracy. However, it has to take in account all the flows, and usually a data stream contains more than 1.5M flows. Furthermore, this method does not immediately provides the sorted list of top-k on the flow cardinality, but a specific computation must be done for each monitored flow to estimate its cardinality. A recent work that supersedes this limitation is SpreadSketch [13]. This sketch data structure is invertible, since it stores in each bucket also the key of the dominant flow. This leads to extract the identification of superspreaders from the sketch at the end of an epoch. However, this data structure still suffers the limitation of sharing counters across all flows in the stream.

Other work, such as [2,3], are based on sampling to identify flows that exceed a fixed threshold according to the cardinality. However, sampling can result in reducing accuracy on small time scales. And, even if [3] further retains all the elements in a suitable data structure (as an hash table) to remove duplicates, this solution has two issues: (i) the fixed threshold is usually do not know a-priori, while it seems more flexible to select the top-k flows regardless to their actual value; (ii) if the cardinality is extremely high, the data structure used to filter duplicates can become extremely big.

3. The FlowFight algorithm

FlowFight is an efficient top-k spreader detection algorithm that uses the HLL sketch to estimate the cardinality of each flow. In fact, the HLL sketch provides a very efficient estimation of the number

of distinct elements in the stream. However, we cannot allocate one HLL for each flow because it will require a huge amount of memory. Hence, the algorithm dynamically selects a subset of flows to monitor as candidate top-k flows. The idea is to assign exclusive sketches responsible to estimate the cardinality for a subset of the flows, and to apply a randomized access policy for the others.

In the rest of this section, we first show the main steps composing the algorithm. And later, we discuss in details the insertion and update operations through the data structure and finally the randomized access policy.

3.1. Algorithm overview

The FlowFight algorithm is composed by several steps that are executed when a packet is processed. These steps are summarized in Fig. 3. FlowFight first parses the packet to extract flow[3] information and identify the item to count. It is worth noticing that we need two different keys from each packet: a *flow_key* and a *discriminator_key*. The *flow_key* is a selection of packet fields that defines the flow the packet belongs, since the algorithm allocates a HLL for each flow. While the *discriminator_key* is a selection of packet fields to discriminate distinct elements (i.e. to count the unique occurrence inside the flow). As an example, the *flow_key* can be the *source_ip*, consequently, all the packets with the same *source_ip* hit the same HLL. And, the *discriminator_key* can be the *destination_ip*, consequently, all the packets with the same *source_ip* and *destination_ip* are counted just once. The choice on the subset of packet fields for flow and discriminator keys depends on the use case interested. In case of DDoS detection, the task is to detect hosts that try to connect to multiple destinations: the flow key can be set as a *source_ip* and the discriminator key as a *destination_ip*. Another example is a load balancing application in which the number of unique connections is balanced among different servers: the flow key can be set as a *destination_ip* and the discriminator key as the 5-tuple. A further example could be the need to adjust the capacity of the front-end system to the number of HTTP requests: the flow key can be set as a 5-tuple and the discriminator key as HTTP GET request.

The second step queries the data structure to identify if the ingress flow is a candidate flow (i.e. if the flow belongs to the monitored flows inside the data structure). While, not monitored flows are analyzed by the randomized access policy to decide if admit or ignore the flow. We propose an innovative randomized access policy based on a "fight" between flows (details in Section 3.3). The idea is to use the HLL rank value (i.e. the position of the leftmost 1 of the hash) as the discriminator factor to accept new flows against the smallest monitored flow. This comes to the fact that the HLL rank value provides an initial evaluation of the flow: as an example, if the rank value of a packet is 10, it can be considered as a representative of a flow that can contain 2^{10} distinct elements.

Thus, in the randomized access policy, if the ingress flow wins the fight, it becomes a candidate flow and it will be inserted in the data structure at the expense of the smallest monitored flow. However, the ingress flow will not become the new smallest one but it will be initially placed in a "promotion zone": a safe zone in which time to grow is given to just admitted flows before let them fight to survive. This is needed since the HLL gives good results when has encountered several packets. Instead, if the ingress flow loses the fight, the algorithm moves to the next ingress packet.

For the candidate flows (or the just admitted ones), the algorithm updates the HLL sketch associated, and computes a rough estimation of the cardinality from the HLL sketch. This rough estimation is needed to quantify monitored flows and sort them in the data structure. The FlowFight algorithm needs the data structure sorted in order to quickly detect the smallest monitored flow. It is worth noticing that the regular HLL estimation is computationally complex to process real-time, i.e. it requires to compute the harmonic mean over all the registers ($reg[j]$) of the HLL sketch ($E = \alpha_m \cdot m^2 \cdot Z^{-1}$, where $Z = \sum_{j=1}^{m} 2^{-reg[j]}$). So, to speedup these operations, our strategy is to implement a rough runtime approximation (i.e. just compute the sum of registers) and use it as a discriminating factor to identify the flows with the largest number of distinct elements. The data structure and its update operations are discussed in Section 3.2.

As an example of packet processing by FlowFight, Fig. 4 shows two cases in which the FlowFight algorithm receives first a candidate flow and then a not monitored flow. We suppose to have infinite memory to represent all the flows in the data stream. However, for the not monitored flows, the table does not report information. Each entry in the table reports the flow id, the rough estimation, the register counter and an HLL sketch of four registers.

The first case in Fig. 4(a) shows an ingress packet that belongs to the flow with id = 7, whose HLL rank is 8, to be counted on the fourth register. After updating the sketch, the rough estimation ($2^3 + 2^6 + 2^2 + 2^5 = 108$) is updated by simply removing the old register value with the new one ($108 - 2^5 + 2^8 = 332$). And since, the rough estimation is higher than the one of the adjacent flow, it performs a switch between these two entries to maintain the table sorted. Instead, Fig. 4(b) shows a case of a not monitored flow. The ingress packet belongs to the flow with id = 71. Thus, FlowFight applies the randomized access policy (i.e. ①): the ingress packet has a HLL rank = 9 and its rough estimation (i.e. $2^9 = 512$) is higher than the smallest flow in the data structure (i.e. 80), consequently winning the fight. The smallest flow is kicked out (i.e. ②). The ingress flow is placed in the promotion zone and one flow in the promotion zone becomes the new opponent flow (i.e. ③).

Finally, to report the top-k flows, the algorithm computes the regular HLL estimation for all the monitored flows. It sorts the list according to this more accurate estimation and returns just the first-k flows.

3.2. Identify, update and sort candidate flows

The algorithm needs a data structure that *cheaply* identifies monitored flow, updates the HLL sketch for each flow and at the same time easily detects the flow with the lowest cardinality to kick out. An interesting candidate seems to be the well known *Stream-Summary* data structure [11,24], since it provides $O(1)$ overhead to search or update a flow, or find the smallest flow. However, it is tailored to the flow size case in which a single counter is needed. Therefore, we re-design it to accommodate several sketches and manage them. As in Fig. 5, the data structure uses a hash table to minimize the cost to access data in updating the sketch. This table is accessed by a flow key and returns two indexes, one used to access the HLL table and the other one to access the flow state table. The HLL table contains a sketch in each row and it is not sorted. While, the flow state table contains additional information to keep track of a rough estimation of the cardinality and indexes needed in the sorting phase.

As proposed by Compact Space Saving (CSS) [24], our data structure uses array indexes instead of memory pointers as in SS [11]. In this way, the data structure works on statically allocated memory resulting also suitable for hardware implementations. However, CSS uses an intermediate table, called value-index, to sort flows. The authors propose to use a mapping between the flow size value and the index of the largest counter index with the same value. Consequently, the algorithm can sort in a faster way by just swapping the updated counter with the counter found in the value-index table. However, this approach has two limitations: (i) it works well when the single counter is only updated incrementally and, (ii) accessing this additional table is not trivial.

Therefore, we design an ad-hoc solution to sort the flow state table. We decided to perform the sorting with a swap procedure between a flow and its adjacent entry. In particular, when a flow updates its

[3] We used the term "flow" following the OpenFlow [26] terminology in which a flow is defined by all packets matching a generic rule, such as (src_ip=10.0.0.1, dst_ip=*, src_port=*,dst_port=*,).

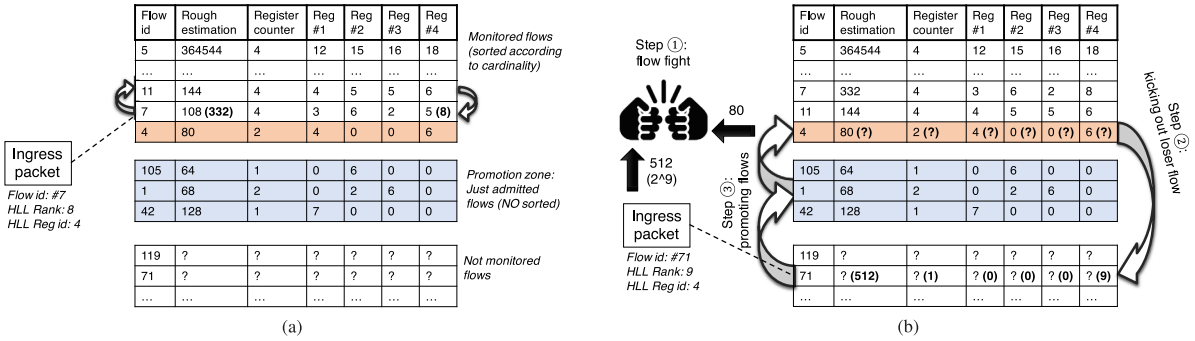


Fig. 4. FlowFight packet processing. Case-ⓐ: the ingress packet belongs to a candidate flow. FlowFight updates the register in the sketch and the rough estimation. Then, it switches the ingress flow with the adjacent flow to maintain table sorted with a swap operation. Case-ⓑ: the ingress packet belongs to a not monitored flow. Step-①: the ingress flow fights to enter against the opponent flow, winning the fight. Step-②: the opponent flow is kicked out and its information are discarded. Step-③: the ingress flow is promoted, and one flow randomly chosen in the promotion zone becomes the new opponent flow. Updates are reported in bold.

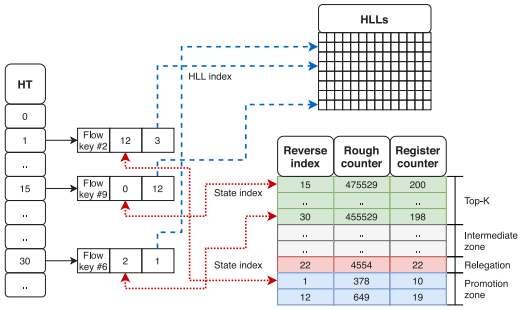


Fig. 5. Stream summary re-design for top-k estimation with HLL.

own rough counter, it will check the adjacent entry in the flow state table, and if this has a smaller counter value it will perform the swap between these two entries. This approach is less demanding than the CSS, but it is worse in terms of convergence time, i.e. the data structure is not perfectly sorted according to the cardinality estimation. To overcome this limitation, we postpone the actual ordering and the final decision about the selection of the top-k to the query phase, in which we can: (i) perform a less approximate evaluation using the standard harmonic mean instead of the rough estimation, and (ii) consider all the monitored flows for the sorting procedure.

Rough estimation of the HLL sketch. We explored two different approaches to perform a fast and approximate estimation of a single HLL: the former uses a less accurate estimation that can be performed with few arithmetic operations; while the latter is based on the streaming HLL method proposed by Ting in [20], which is more accurate, but it involves more complex arithmetic operations (e.g. floating point divisions).

The first method performs the sum of the registers in the sketch. This value is less robust than the harmonic mean. But, it is easier to compute in real time, since it does not involve divisions. Moreover, this value can be computed in a smarter and recursive way: when a HLL register is updated by a packet arrival, rather than compute $\tilde{E}$ from scratch, the algorithm just removes the old register value and add the new one on $\tilde{E}$ already computed. The proposed method is reported in Eq. (1), where i is the register index to update. To notice that the method requires just to perform few shifts and add operations and it is therefore suitable for an high speed implementation.

$$\tilde{E}_{new} = \tilde{E}_{old} - 2^{reg[i]_{old}} + 2^{reg[i]_{new}} \tag{1}$$

The second method is proposed by Ting in [20] that also provides a runtime estimation of the cardinality, which is much more accurate than the first one (and also of the regular harmonic mean estimation). Unfortunately, as shown in Section 5, this approach has processing requirements that can limit the overall throughput of the top-k spreader detection. Furthermore, the use of floating point operations makes this approach unfeasible for a pure hardware implementation of the FlowFight algorithm. The cardinality estimation is reported in Eq. (2), where q is the probability that the sketch is modified.

$$q_{new} = q_{old} + 2^{-reg[i]_{old}} - 2^{-reg[i]_{new}} \tag{2}$$

$$\tilde{E}_{new} = \tilde{E}_{old} + 1/q_{old} \tag{3}$$

3.3. The randomized access policy

Since FlowFight monitors just a subset of flows in the data stream, it needs a randomized access policy to dynamically update flows inside the data structure. The FlowFight randomized access policy can be thought as a fight between two flows, as depicted in Fig. 6. The former is the smallest flow in the data structure, called opponent flow, that is a monitored flow and of which we have some information (e.g. a not empty sketch and a rough estimation). The latter is the ingress flow that is not monitored and we have no information.

The idea is to use a bias coin flip (i.e. the HLL rank value, the position of the leftmost 1 of the hash) as the discriminator factor to admit new flows. For the opponent flow, the rough estimation of the flow is already computed and stored in the data structure. While for the ingress flow, the HLL rank is weighted according to the number of registers different from zero in the opponent sketch. In this way, the new flow will be approximately estimated as an empty HLL in which a number of registers (a fraction of the registers of the opponent flow) will have the rank just computed.

This choice gives to new flows a chance to fairly fight against opponent flows that present high rough estimations because they have many registers different from zero. However, the side effect of this choice can be that outlier flows (i.e. flows with high rank in few registers) can enter and stack the policy. To limit this side effect, an additional check to the opponent flow is performed according to the number of registers updated. In fact, if a flow fills few registers even with high values, it is not valued by the original HLL that uses in these cases a linear counting on the number of registers for the final estimation. As will be shown in Section 4.1, the combined use of the rough estimation and of the number of non zero registers is needed to reduce the approximation error and the presence of outliers, greatly improving the capacity of the system to identify the candidate new flows to monitor. On the other hand, this approach requires to use some thresholds to correctly weight the two values. However, during our simulations we found that the threshold values are not critical, since also significant variation of these threshold does not affect significantly the final results.

Moreover, the randomized access policy presents an initial filter to the ingress flow according to a minimum value of rank. This is

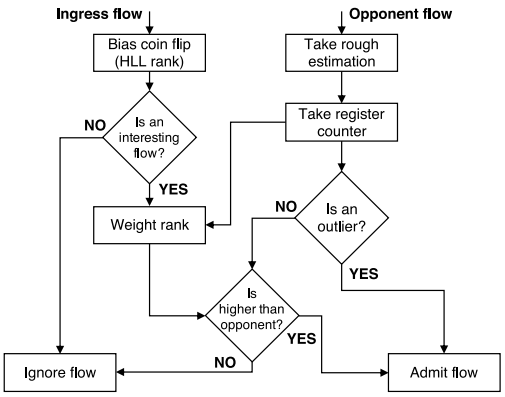


Fig. 6. The FlowFight randomized access policy: a fight between the ingress flow and the opponent flow (i.e. the smallest flow in the data structure).

more an optimization to boost the performance than a fundamental part of the strategy. In fact, let only bigger flows play the game can save clock cycles: e.g. packets with $rank = \{1, 2, 3, 4\}$ are respectively representative of $\{2, 4, 8, 16\}$ distinct elements and have less chances to win.

Switching candidate flows after admission. When the ingress flow wins the battle, it is admitted by the policy and the loser flow is kicked out. Since the data structure has a fixed size, each time a new flow is admitted another one must be evicted. The loser flow is the best candidate because it already spent enough time in the promotion zone, but due to its relatively low estimation it has not been able to survive in the fights.

At this point, a new HLL sketch is now associated to the ingress flow and the flow is placed in the promotion zone. The promotion zone is a set of flows that have just been admitted. This zone can be thought as a safe zone in which time to grow is given to flows before let them fight in the relegation spot. This is needed since the HLL gives good results when has encountered several packets. In fact, the ingress flow cannot inherit the estimation from the deleted flow as usually done in traditional top-k algorithm. This approach is unfeasible for sketches. Sketch is an array of counters that are updated independently each other. And the way in which each counter is updated depends on the statistical properties of the single flow. As an example, let suppose that two flows update simultaneously the same HLL sketch: (i) if the two flows contain the same number of distinct elements but with different elements, the resulted estimation from the sketch is the sum of the cardinality of the two flows; (ii) while if the two flows contain the same number of distinct elements and the same elements, the resulted estimation is the cardinality of the single flow.

Finally, when the ingress flow is admitted and placed in the promotion zone, there is a "promotion" of one of the existing flows as the new opponent flow for next ingress packets.

4. Algorithm assessment

Approximate estimation methods introduce inaccuracy in the value they compute. Consequently, this error will be propagated when integrating these methods into complex algorithms, such as the top-k estimation algorithm proposed in this paper. In particular, this error can be propagated on the top-k precision metric, which corresponds to the ratio of the number of correctly estimated flows by the algorithm on the top-k size.

As an example, let suppose that we have infinite memory and we can store all the flows of the stream in our data structure. Since the data structure sorts flows according to the estimation of their cardinality (and not the exact value), the first k flows are the ones with the highest estimation value. However, the HLL sketch estimates cardinality with

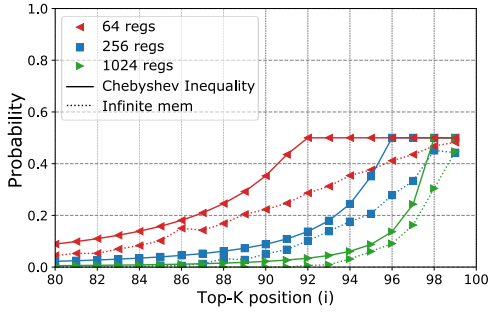


Fig. 7. Intrinsic limitation in top-k precision.

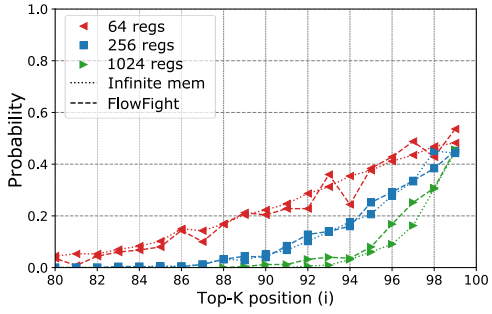


Fig. 8. Algorithm assessment: Infinite memory vs FlowFight.

an error as reported in Eq. (4). Thus, if we use a sketch of $m = 512$ registers (≈ 256 bytes), the cardinality value of each flow is affected by a Relative Standard Error of 5%. The standard error can be computed as:

$$\sigma_{hll} = \frac{1.04}{\sqrt{m}} \cdot C(f) \tag{4}$$

Let us now suppose the stream contains F flows whose cardinality shapes a zipf distribution as in Eq. (5).

$$C(f_i) = \frac{K}{i}, \forall i \in \{1, \dots, F\} \tag{5}$$

And, let assign an HLL sketch for each flow, and define $\tilde{C}(f_i)$ as the estimation of its cardinality from the sketch. We are interested in the probability that a certain flow swaps the position with the last element in the top-k list. This means that its estimation is lower than the exact value of the last flow in top-k. Thus, we can use the Chebyshev inequality in the symmetric case as in Eq. (6).

$$\begin{aligned} P\left(|\tilde{C}(f_i) - C(f_i)| > \lambda \cdot \sigma_{hll}\right) &\leq \frac{1}{\lambda^2} \\ P\left(\tilde{C}(f_i) < C(f_i) - \lambda \cdot \sigma_{hll}\right) &\leq \frac{1}{2 \cdot \lambda^2} \end{aligned} \tag{6}$$

And, our case can be defined as the event in which the estimation of flow i is lower of the exact value of flow x (i.e. $\tilde{C}(f_i) < C(f_x)$), where $C(f_i) > C(f_x)$. This refers to the λ value reported in Eq. (7).

$$\begin{aligned} &\tilde{C}(f_i) < C(f_i) - \lambda \cdot \sigma_{hll} = C(f_x) \\ &\frac{K}{i} - \lambda \cdot \sigma_{hll} = \frac{K}{x} \\ &\lambda = \frac{K \cdot (x - i)}{x \cdot i \cdot \sigma_{hll}} \end{aligned} \tag{7}$$

Then, the probability that the estimation of flow i is lower than the exact value of flow x is bounded as in Eq. (8).

$$P\left(\tilde{C}(f_i) < C(f_x)\right) \leq \frac{1}{2 \cdot \lambda^2} = \frac{1}{2} \cdot \left(\frac{\frac{1.04}{\sqrt{m}} \cdot x}{(x - i)}\right)^2 \tag{8}$$

Eq. (8) approximates the actual error probability since compares the estimated value of $\tilde{C}(f_i)$ with the exact value of $C(f_x)$. A more detailed model should consider also the estimated value $\tilde{C}(f_x)$. However, the simulations we performed show that this is significant only when the number of HLL registers is small. Therefore the extension of the bound is left as future work.

Focusing on the case in which $x = 100$ and $i < 100$, Fig. 7 reports Eq. (8) when different sizes of HLL are used (i.e. {64, 256, 1024} regs) that correspond to different estimation errors (i.e. respectively {13%, 6.5%, 3.25%}). Moreover, the figure shows the distance between the Chebyshev inequality to respect simulation results in which we allocate an HLL for each flow and we order them according to the HLL estimation ("Infinite mem"). The test has been performed using a synthetic trace shaped by a Zipf distribution with $\alpha = 1.1$ (to reproduce Eq. (5)) and iterating result over 400 trials. As already mentioned, the plot shows that Eq. (8) is a weak bound when the number of HLL registers is low, but becomes significantly close to the actual error probability when the number of HLL registers grows.

While, Fig. 8 analyzes the impact of the proposed FlowFight algorithm. The results show the gap between the ideal case in which we allocate an HLL for each flow and the proposed FlowFight algorithm in which a subset of flows are selected. It is worth noticing that the difference between the two cases are negligible.

Finally, we can extend the considerations done for the Zipf distribution to a generic distribution, as in Eq. (9).

$$\begin{aligned} & C(f_i) - \lambda \cdot \sigma_{hll} = C(f_x) \\ & \lambda = \frac{C(f_i) - C(f_x)}{\sigma_{hll}} \\ & P\left(\tilde{C}(f_i) < C(f_x)\right) \leq \frac{1}{2 \cdot \lambda^2} = \frac{1}{2} \cdot \left(\frac{\frac{1.04}{\sqrt{m}} \cdot C(f_i)}{C(f_i) - C(f_x)} \right)^2 \end{aligned} \tag{9}$$

As an example consider Fig. 2 in which the cardinality distribution per flow of a CAIDA trace is shown. The situation is exactly what we just described. There is a crowded boundary around the 100th flow (when the estimation is affected by a 5% error on the exact value). And, due to the estimation error, the flows between the 95th-107th index can be interchangeable each other.

4.1. Microbench analysis

As already discussed in Section 3.3, the randomized access policy plays a key role in the FlowFight algorithm, thus we investigate the impact of each step of the policy to assess the algorithm. We develop a software simulator able to process packets from a trace file. In this way, the simulator can generate golden results and exactly counts size and cardinality of all the flows composing the trace. On the other side, the simulator leads to estimate the overall accuracy of the developed algorithm and the impact of the different steps that compose the algorithm. This latter analysis has been carried out developing several microbenchmarks that permits to investigate the impact of some algorithm parameters on the results quality. The test has been performed using a synthetic trace shaped by a Zipf distribution with $\alpha = 1.1$. We deployed HLL sketches with 256 registers, which is a configuration widely used for cardinality estimation in networking scenarios.

We analyzed three different scenarios: Fig. 9(a) shows a complete overview in which the different steps of the randomized access policy are tested separately; Fig. 9(b) focuses on the admission policy rule defined for the ingress and opponent flows; Fig. 9(c) analyzes the impact of a guard interval (i.e. "Intermediate zone") to reduce the gap between FlowFight and the ideal case with infinite memory discussed in the previous section.

Fig. 9(a) shows that each step of the algorithm cannot achieve acceptable performance since they are below 80% of precision. About weighting the estimation of the ingress flow: "OnlyMatch" corresponds to the case in which we do not scale the incoming flow, while "WeightedMatch" to the case in which we weighted the rough estimation of the new flow with the number of non zero elements of the opponent. The first option privileges the opponent flow, while the second privileges the incoming flow. To mitigate outliers effect, we further consider the case in which we check that registers different from zero of the opponent flow ("CheckRegs") is below a threshold and a case in which we apply an expiration timer that kick out flows that are not reached a minimum number of non zero elements after an interval of packet processed ("PktsMargin", set to 6400 pkts). All these cases are not able to achieve a precision higher than 80%. Hence, we now focus on the deployment of more HLL sketches in order to include flows that will be not counted on the final top-k, such as allocating the promotion zone ("Promotion zone") and the intermediate zone ("Intermediate zone"). The promotion zone is a set of flows just admitted, while the intermediate zone is a set of survived flows but not enough big. It is worth noticing that this strategy produces higher benefits to respect the use of a timer of packet processed to let HLL sketch be populated. In particular, "Intermediate zone"-line represents the FlowFight as a whole, and it shows that the use of both the zones allows to reach an accuracy of more than 95% and a smaller std as shown by boxes in figure.

While in Fig. 9(b), we test different thresholds for the packet timer and the weighted match (i.e. the minimum number of register different from zeros in the opponent flow to increment weights). In this case, it is worth notice that a good trade-off for the two thresholds seems to be 6400 pkts for the "PktsMargin" and 64 registers different from zeros for the "WeightedMatch". The experimental results reported in Section 5 confirm that these parameters values give good results with different tracesets. However, the behavior of the FlowFight algorithm seems quite robust with respect to the choice of these parameters. In fact, Fig. 9(b) shows that the use of different parameter values impact on the overall precision for less than 5%, std remains in line as in Fig. 9(a).

Finally, Fig. 9(c) analyzes the memory needed in the intermediate zone to improve accuracy and to reach similar performance to the ideal case. We tested different configurations (i.e. from 1 to 256 additional HLLs in the intermediate zone) and we report the first index in the top-k that has the probability greater than 0.4 to swap its position with the 100th flow. This test has been performed iterating results over 400 trials. Results show that allocating more than 64 additional HLLs impact on the overall precision for less than 5%.

5. Experimental results

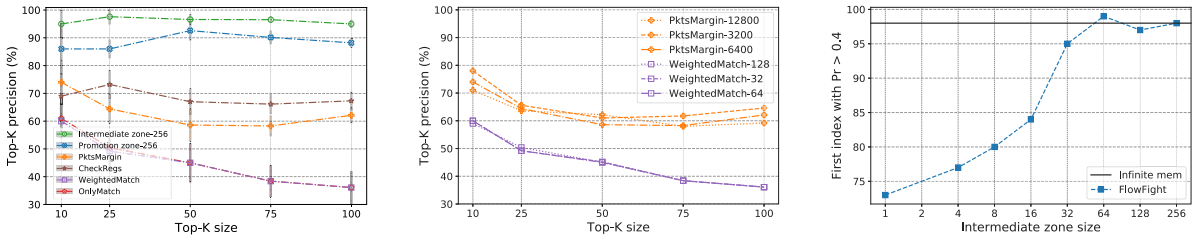
5.1. Methodology

In this section, we outline the methodology we used in our experimental campaign, which allows to get realistic yet repeatable[4] benchmark results, which comprises several elements that are illustrated in Fig. 10: namely, the Synthetic Stream Generator (SSG), a programmable packet generator, such as MoonGen, and the Device Under Test (DUT).

Testbed setup. Our testbed setup consists in a COTS server with 2× Intel Xeon Silver 4110, each with 16 physical cores running at 2.10 GHz with 128 GB RAM memory. The server is equipped with 2× Intel X520 NICs with dual-port 10 Gbps full duplex links directly connected with SFP+ interfaces.

Synthetic Stream Generator and CAIDA traces. We first use the Synthetic Stream Generator (SSG) to generate a sequences of synthetic packet headers. This representation is compliant with the ClassBench [27] format and allows to easily shape the final traffic with

[4] Our implementation and test environment is available at GitHub: https://github.com/valebru/FlowFight-on-VPP.



(a) FlowFight analysis. (b) Admission policy analysis. (c) Memory analysis.

Fig. 9. Algorithm assessment: microbench analysis.

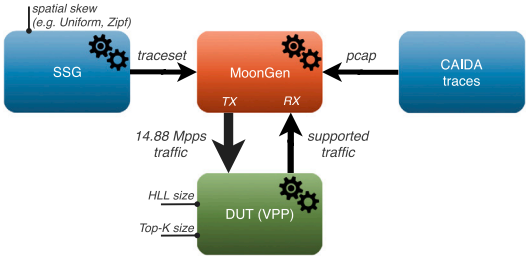


Fig. 10. Synoptic of experimental methodology.

Table 1
Experimental workloads.

Type	Pkts	Flows	Cardinality		Dup
			Range	Avg	
uniform	320k–6.4M	10–100	32k–64k	48k	NO
zipf (1.1)	6M–7M	1.9M	1–616k	3.41	NO
zipf (1.4)	6M–7M	103k	1–2M	63.02	NO
zipf (1.8)	6M–7M	8.5k	1–3.4M	758.61	NO
Caida2015	22M	210k	1–23k	4.26	YES
Caida2019	76M	1M	1–345k	5.38	YES

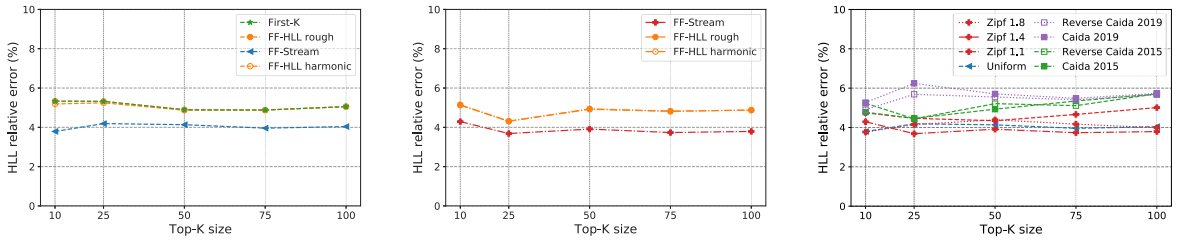
the desired stream distribution in terms of {flows X distinct elements}. For our experimental campaign, we focus on detecting the top-k flows (defined by the *destination_ip*) with the largest cardinality (defined by the 5-tuple). We generate two different kind of synthetic datasets. The former is generated by shaping an uniform distribution: all the flows (10–100) have the same number of distinct elements (32k-64k); this kind of dataset allows to test the impact of the update operations in the algorithm without involving the randomized access policy, because the number of flows are limited and can be monitored as a whole. The latter is generated by shaping a zipf distribution [28,29] (with $\alpha = \{1.1, 1.4, 1.8\}$): these synthetic datasets likely match real pattern in realistic traces, but they are more stressful since they contain more flows and a vaster range of distinct elements. This dataset has the property of not showing repeated elements. Finally, both the workloads are replicated and shuffled ten times to test the algorithm robustness to the stream order.

To further provide realistic results, we also consider real traces from the CAIDA [30] dataset (we use five traces from Caida 2015 and five traces from Caida 2019). Packets are cropped to 64B to emulate a worst-case scenario in terms of packet processing throughput. This workload is similar to the one shaped by the zipf distribution, but the elements can be repeated (i.e. there are multiple packets with the same 5-tuple). The presence of duplicated elements affects the systems both with respect to the algorithm accuracy and with respect to the packet processing throughput. Duplication impacts accuracy since the randomized access policy is triggered multiple times from duplicated packets. Therefore the incoming flow has multiple chances to be admitted, by fighting against different opponents. While from the packet processing throughput point of view, it must be noticed that flows with an high number of duplicates can enter in the CPU cache increasing the processing speed for these flows. Moreover, we replicate the CAIDA datasets in reverse mode to test, even in this case, the algorithm robustness to the stream order. Finally, the workloads are summarized in Table 1, in which difference between each others are outlined.

Workload generation (MoonGen). The generation of a stream with a specific (and exact) sequence of distinct elements is not trivial. In fact, we experienced several errors in the generation of packets to respect the expected ones. This is because the traffic generator cannot generate different packets at a high transmission rate without exploiting a cache. This is an issue that can affect the detection and the evaluation of performance metrics such as top-k precision. Therefore, we care to develop a solution that can guarantee this requirement. The proposed solution is a MoonGen [31] application that takes the traceset as input, it first generates the entire stream in memory and then transmit packets in a 10 Gbps stream of 64 Bytes packets as output. Thus, our solution results able to generate a worst-case rate of 14.88 Mpps without altering the expected number of distinct elements. We further ensure each packet payload has the minimum size of 64 Bytes to stress the packet classifier. Considering bytes of Ethernet preamble and 9.6 ns of inter-frame gap, this leads to a worst-case rate of 14.88 Mpps.

Device Under Test (VPP). As candidate full-blown software processing framework to embed the above algorithm, we consider a popular open-source framework, namely Vector Packet Processor (VPP) [12,32] as the DUT. VPP is released in the context of the Linux Foundation's Fast Data IO (FD.io) project [33] and it is designed to take advantage of general-purpose CPU architectures and implements a full network stack in user-space leveraging Data Plane Development Kit (DPDK) [34]. The main feature is the ability by VPP to share tasks overhead by processing packets in *vectors* [35]: once a batch of packets is grabbed from the NIC, the code is applied to all packets in the batch, by additionally leveraging pre-fetching to fully utilize CPUs without being stalled by memory access.

Therefore, in our experimental methodology, we instantiate our algorithm in the DUT (i.e, VPP), on a single core, that we stress with the workload generated by MoonGen, measuring its performance in terms of packets processed per second, top-k precision and HLL relative error. We developed different versions of the proposed algorithm in VPP: (i) a "First-K" version in which the first k flows are admitted and monitored while the others are not monitored, and in which swaps are not performed in the stream summary data structure (i.e. *First-k*), we will use this implementation to investigate the impact of the HLL update operations on the framework; (ii) a top-k version in which we use the HLL sketch to estimate cardinality, by computing the harmonic mean to sort flows (i.e. *FF-HLL harmonic*) or (iii) exploiting a rough runtime estimation to sort flows (i.e. *FF-HLL rough*), we will use these implementations to compare the impact on the framework of the real HLL estimation and the proposed rough method to sort flows on the fly; (iv) a top-k version in which we use the HLL streaming [20] as a sketch to estimate cardinality (i.e. *FF-Stream*), this version achieves better accuracy in the cardinality estimation at the expense of using complex

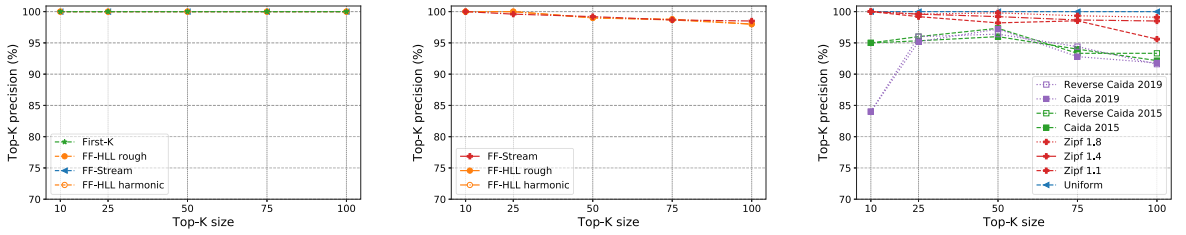


(a) Analysis of different VPP plugin versions applying the uniform workload. (First-k, FF-HLL overlap)

(b) Analysis of different VPP plugin versions applying the zipf 1.4 workload.

(c) Analysis of FF-Stream applying the different workloads.

Fig. 11. HLL accuracy analysis.

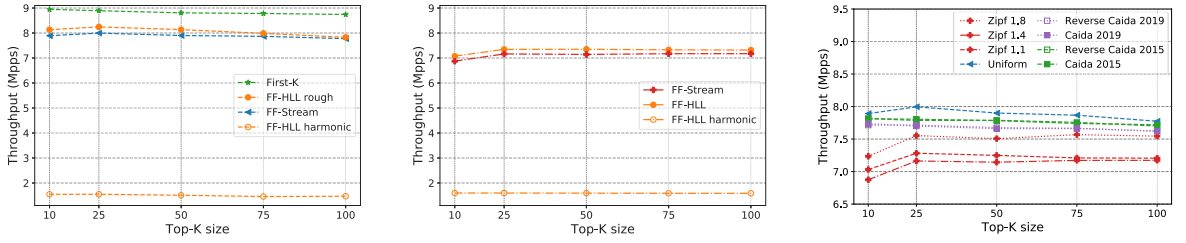


(a) Analysis of different VPP plugin versions applying the uniform workload. (All lines overlap)

(b) Analysis of different VPP plugin versions applying the zipf 1.4 workload.

(c) Analysis of FF-Stream applying the different workloads.

Fig. 12. Top-k precision analysis.



(a) Analysis of different VPP plugin versions applying the uniform workload.

(b) Analysis of different VPP plugin versions applying the zipf 1.4 workload.

(c) Analysis of FF-Stream applying the different workloads.

Fig. 13. Throughput analysis.

update operations. To notice that among different VPP implementations used in the experimental campaign, only *FF-HLL rough* and *FF-Stream* reflect the proposed algorithm described in previous sections.

Finally, accordingly to algorithm assessment in Section 4.1, the experiment has been performed allocating 256 entries for the promotion and the intermediate zone, with each HLL composed of 256 registers. Experimental results are reported as average over the different runs, while standard deviation is reported in round brackets.

5.2. HLL accuracy analysis

In this section, we investigate sketch accuracy in terms of cardinality estimation of the single flow. In particular, we analyze the two estimation methods described above: the standard HLL and the streaming one.

We first perform a basic experiment in which we allocate an HLL for each ingress flow in the dataset by applying the uniform workload. This kind of experiment will give us the real accuracy of the HLL sketch and also the difference between the original HLL version and the HLL streaming proposed by [20]. In Fig. 11(a), the HLL relative error is reported. In this configuration, the HLL processes all distinct elements inside each flow. In fact, all the flows are monitored simultaneously. As expected, the HLL streaming sketch achieves a smaller relative error than the original one: the standard HLL (i.e. *First-K* and *FF-HLL*) achieves a relative error in the order of 5.04% (±0.22%), while the HLL streaming (i.e. *FF-Stream*) achieves a relative error in the order of 4.02% (±0.43%). We now apply a more realistic workload shaped by the zipf distribution: even in this more realistic scenario, results in Fig. 11(b) remain in line with the previous experiment highlighting that HLL streaming has a smaller relative error in the cardinality estimation. While Fig. 11(c) reports results obtained by *FF-Stream* when all the workloads of the methodology are applied.

In summary, this experiment has shown that HLL streaming achieves better accuracy to respect the traditional HLL: this can be exploited to correctly detect and sort the flows by FlowFight, consequently improving the top-k precision in a runtime approach.

5.3. Top-k precision analysis

We now focus on the precision accuracy achieved by the algorithm. Top-k precision is the ratio of the number of correctly estimated flows by the algorithm on the top-k size. To discriminate flows that correctly belong to the top-k, we use golden results computed by the simulator on each workload and we compare them with the experimental results obtained by the DUT.

We first investigate a basic configuration in which we can allocate an HLL for each flow in the dataset by applying the uniform workload. This simple configuration shows FlowFight correctly estimates all the flows for all the cases considered as shown in Fig. 12(a). However, this experiment has shown a case in which the randomized access policy is not stressed. Therefore, Fig. 12(b) shows the case in which the zipf 1.4 workload is applied to *FF-HLL* and *FF-Stream* VPP implementation. This

kind of experiment reproduces a more realistic scenario. The results show that both the implementations are able to correctly estimate more than 95% (with std around 1%) of the flows in the top-k. Moreover, it can be seen that accuracy decreases when the top-k size increases. This is due to the intrinsic limitation of the top-k precision as discussed in Section 4.

Let us now focus on a single implementation, the *FF-Stream* VPP implementation. And we now apply all the workloads in the methodology. This kind of test will show the real performance of the algorithm in terms of precision as the number of flows correctly estimated over the real top flows of several datasets. In Fig. 12(c), precision accuracy is reported. The uniform workload as before trivially estimates all the flows of the dataset. The Zipf workloads estimate correctly about 95%–99% of the exact top flows of the original dataset (with std less than 1%). Finally, Caida datasets are in line with synthetic ones achieving a precision of 95% (±2.31%), except for the case in which we estimate the top 10 flows where std increase up to ±10%. For this particular case, we discovered that the cardinality difference between the 8th and the 10th flow are smaller than the HLL precision in some Caida-19 traces and therefore the detection can fail. This is the same condition discussed in Fig. 2. Thus the lower precision is not due to the FlowFight algorithm, but to the limitation of the HLL estimation.

In summary, this experiment has shown VPP implementations achieving more than 95% of accuracy, like results on the ideal case in which an HLL is deployed for each flow and discussed in Section 4. Moreover, it can be noticed that *FF-HLL rough* and *FF-Stream* achieve similar results.

5.4. Throughput analysis

In this section, we focus on an important metric when dealing with packet processing frameworks: throughput as the number of packet processed per seconds. This metric gives the real impact inside the framework and the possibility to be integrated in real devices.

In Fig. 13(a), we apply the uniform workload and we instantiate all the VPP implementations. The figure shows three degradation in performance for three different approaches used in the code. The first one is the degradation between the basic case in which we do not perform swaps in updating the data structure and the regular algorithm (i.e. *First-K* vs *FF-HLL rough*). So the difference between these two lines represent the impact of sorting and memory swaps. *First-K* achieves a throughput of 8.93 Mpps (±0.028), while *FF-HLL rough* achieves 8.165 Mpps (±0.09). Then, the second degradation is caused by the used of division between floating points in the dataplane to support the HLL streaming sketch (i.e *FF-Stream*). However, *FF-Stream* and *FF-HLL rough* converges to similar results when the top-k size increases. In average, *FF-Stream* achieves 7.7158 Mpps (±0.04) compared to 8.165 Mpps of *FF-HLL rough*. The last degradation is related to the harmonic mean used to sort the data structure (i.e. *FF-HLL harmonic*). The harmonic mean is very demanding and this saturates the VPP packet processing allowing to reach just a rate of 1.5076 Mpps (±0.018). Therefore, this strategy cannot be applied in the dataplane, since using an approximate runtime estimation (i.e. *FF-Stream* and *FF-HLL rough*) achieves much higher performances.

In Fig. 13(b), we perform the experiment for a more realistic case exploiting the zipf 1.4 workload and stressing the randomized access policy. These results show that the gap between the two implementations remains in line with the previous experiment, less than 0.5 Mpps. While both the lines are shifted down to respect the uniform case, since in this experiment we further test the randomized access policy, that inserts and deletes elements in the HT, the most computational expensive phase of the algorithm. Both *FF-Stream* and *FF-HLL rough* are above 7 Mpps, while *FF-HLL harmonic* remains an unfeasible solution.

Fig. 13(c) shows the results when we applied several workloads to *FF-Stream* VPP implementation. The uniform workload is the baseline of the experiment. Caida workloads perform better than others probably due to duplicated elements in the flow exploiting cache benefits. While the zipf workloads represent the worst case in which there are the highest number of flows that are admitted and consequently there are more inserted/deleted flows in the data structure. To notice that for all the experiments, the throughput remains constants for all the workloads when the top-k size increases. Finally, we further investigated the number of cache misses with the linux perf tool on the different VPP-FF implementations. As shown in Fig. 14, the results confirmed that the benefits of FF-Stream and FF-HLL rough are mainly due to the minor number of cache misses.

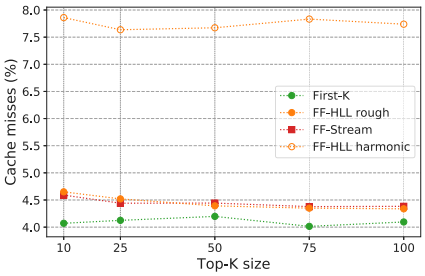


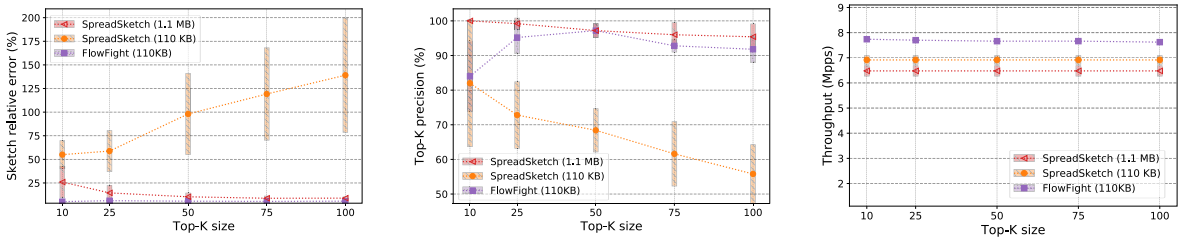
Fig. 14. Cache misses analysis of different VPP plugin versions applying the Caida 2019 workload.

5.5. Comparison with spreadsketch

As introduced in Section 2, a recent algorithm suitable to perform top-k spreader detection is SpreadSketch [13]. In this subsection, we investigated SpreadSketch accuracy in terms of top-k precision and cardinality estimation relative error and compare it against our proposed approach. To make a fair comparison, we proposed two SpreadSketch configurations according to its memory occupancy: 4 rows and 512 columns, which corresponds to 110 KB; and, 4 rows and 4096 columns, which corresponds to 1.1 MB. The first configuration has the same memory occupancy of FlowFight used in our experimental campaign while the second configuration use 10x times more memory. It is worth noticing that we are conservative in estimating FlowFight memory occupancy, because traditional HLL maps a register on 5 bits (as in FlowFight), while more advanced HLL versions (i.e. [10]) map a register on 3 bits without affecting the accuracy (which corresponds to 70 KB).

Fig. 15(b) reports the top-k precision analysis. SpreadSketch is not able to achieve same accuracy of FlowFight when both the data structures use the same amount of memory. This configuration of SpreadSketch has higher error in estimating top-k since counters are shared between all the flows inside the stream. This can be seen in Fig. 15(a) where the relative error on the cardinality estimation of the single flow is reported. Fig. 15(a) shows SpreadSketch affecting the estimation of the single flow around +60% and +140% of its original value. This effect can be mitigated allocating more counters, allowing to distribute flows on an higher number of buckets. This reduces the number of collisions on the same bucket, as can be seen for the second configuration of SpreadSketch (i.e. 1.1 MB). This configuration achieves same performance of FlowFight at the expense of using 10x more memory. This memory gap can be incremented by deploying advanced HLL versions for the FlowFight algorithm.

Moreover, we implemented SpreadSketch in VPP to evaluate its processing speed against FlowFight. Results in Fig. 15(c) show FlowFight achieving higher throughput than SpreadSketch. There are three reasons able to explain these results: (i) SpreadSketch monitors all flows and updates the sketch for each packet processed while FlowFight monitors a subset of flow; (ii) SpreadSketch contains multiple rows to be update by a single packet, and each rows is indexed by a different hash (that is the most expensive operation in updating sketch as explained in [36]); (iii) SpreadSketch allocates more memory than FlowFight, and this leads to increase the memory loads/stores and reduce the efficiency in terms of cache hits.



(a) Cardinality estimation relative error analysis. (b) Top-k precision analysis. (c) Throughput analysis: algorithms as a VPP plugin.

Fig. 15. Comparison between FlowFight and SpreadSketch applying the Caida 2019 workload. Boxes show standard deviation between trials.

6. Discussion and future work

In this section, we discuss some practical issues to enhance the proposed solution:

(1) In practice, the number of attacks in the network will vary from time to time. Therefore, it can be interesting to dynamically change the target on the k flows to monitor. However, this can result in costly resizing and resetting of the data structure several times on the fly. Anyway, the FlowFight data structure has already a set of additional flows that can be used to dynamically expand the k selection. And, these additional flows can be queried by user. But, the data structure needs this extra memory as a guard interval to increase top-k precision as discuss in Section 4.1. Thus, these flows can be used to expand the k selection at the expense of top-k precision. And this trade-off led network operator to choose the suitable k according to the specific use case of the monitoring system (e.g. to trigger an alarm for port scan detector, or to identify which hosts require specific traffic engineering policies).

(2) The data structure works by dividing time in epochs: at the beginning of an epoch, the data structure is empty and at the end it reports the top-k flows for that specific epoch, and so on. However, it could be interesting to have an algorithm time-independent able to recycle flows according to their last occurrence. As an example, if the epoch is set to one year, the reported top-k flows could also refer to biggest attacks occurred months ago. While, we would prefer an algorithm able to reproduce the actual picture of the network under analysis. Therefore, as a future work, we are designing a mechanism to let flows decay inside the data structure. This leads active flows to remain inside the data structure, while evicting inactive flows.

7. Conclusions

We presented FlowFight, an algorithm that provides an estimation of the top-k flows with the highest cardinalities. The algorithm has been implemented and tested as a VPP plugin, in this way we integrated our solution in a full-fledged software router. This allowed to test both the accuracy and the processing speed of FlowFight in a real network scenario. The algorithm leverages the HyperLogLog approximate sketch to estimate the cardinality of a single flow. Since the HLL sketch presents both scalability (in terms of memory footprint) and processing (in terms of time to estimate the cardinality using the harmonic mean) issues, we implemented a suitable data structure to monitor multiple flows, a rough cardinality estimation procedure, and a specific randomized access policy to limit the overall memory footprint without sacrificing performance and accuracy. An extensive experimental campaign has been carried out first to identify the algorithm parameters that affects the accuracy and throughput of our algorithm and after to assess its overall performance. The proposed algorithm resulted able to estimate top-k flows with an accuracy of more than 95% and able to reach a throughput around 8 Mpps on single core. Moreover, FlowFight requires 10x times less memory than a state of the art competitor sketch, such as SpreadSketch, to achieve same accuracy results.

CRediT authorship contribution statement

Valerio Bruschi: Conceptualization, Methodology, Software, Writing - Original Draft, Review & editing. **Salvatore Pontarelli:** Conceptualization, Methodology, Validation, Writing - review & editing. **Jerome Tollet:** Conceptualization, Supervision. **Dave Barach:** Conceptualization, Supervision. **Giuseppe Bianchi:** Methodology, Formal analysis, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is partially supported by the Cisco Research Award number 2019-201350 entitled “Stream-Based Monitoring Task at Single and Large Scale in a Fast and Scalable manner”.

References

[1] X. Shi, D.-M. Chiu, J.C. Lui, An online framework for catching top spreaders and scanners, Comput. Netw. 54 (9) (2010) 1375–1388.
[2] Q. Zhao, A. Kumar, J. Xu, Joint data streaming and sampling techniques for detection of super sources and destinations, in: Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, in: IMC ’05, USENIX Association, USA, 2005, p. 7.
[3] S. Venkataraman, D. Song, P.B. Gibbons, A. Blum, New streaming algorithms for fast detection of superspreaders, 2004.
[4] Y. Chabchoub, R. Chiky, B. Dogan, How can sliding hyperloglog and EWMA detect port scan attacks in IP traffic? EURASIP J. Inf. Secur. 2014 (1) (2014) 5.
[5] Y. Liu, W. Chen, Y. Guan, Identifying high-cardinality hosts from network-wide traffic measurements, IEEE Trans. Dependable Secure Comput. 13 (5) (2015) 547–558.
[6] P. Flajolet, G.N. Martin, Probabilistic counting algorithms for data base applications, J. Comput. System Sci. 31 (2) (1985) 182–209.
[7] M. Durand, P. Flajolet, Loglog counting of large cardinalities, in: European Symposium on Algorithms, Springer, 2003, pp. 605–617.
[8] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: Proc. of AOFA, 2007.
[9] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, X. Li, Heavykeeper: An accurate algorithm for finding top- k elephant flows, IEEE/ACM Trans. Netw. 27 (5) (2019) 1845–1858.
[10] Q. Xiao, Y. Zhou, S. Chen, Better with fewer bits: Improving the performance of cardinality estimation of large data streams, in: IEEE INFOCOM 2017-IEEE Conference on Computer Communications, IEEE, 2017, pp. 1–9.
[11] A. Metwally, D. Agrawal, A. El Abbadi, Efficient computation of frequent and top-k elements in data streams, in: International Conference on Database Theory, Springer, 2005, pp. 398–412.
[12] D.R. Barach, E. Dresselhaus, Vectorized software packet forwarding, 2011, US Patent 7, 961, 636.
[13] L. Tang, et al., Spreadsketch: Toward invertible and network-wide detection of superspreaders, in: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 2020, pp. 1608–1617, http://dx.doi.org/10.1109/INFOCOM41043.2020.9155541.
[14] H. Harmouch, F. Naumann, Cardinality estimation: An experimental survey, Proc. VLDB Endowment 11 (4) (2017) 499–512.

[15] A. Metwally, D. Agrawal, A.E. Abbadi, Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic, in: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, 2008, pp. 618–629.

[16] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, 2003, pp. 153–166.

[17] K.-Y. Whang, B.T. Vander-Zanden, H.M. Taylor, A linear-time probabilistic counting algorithm for database applications, ACM Trans. Database Syst. 15 (2) (1990) 208–229.

[18] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting distinct elements in a data stream, in: International Workshop on Randomization and Approximation Techniques in Computer Science, Springer, 2002, pp. 1–10.

[19] K. Beyer, P.J. Haas, B. Reinwald, Y. Sismanis, R. Gemulla, On synopses for distinct-value estimation under multiset operations, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, 2007, pp. 199–210.

[20] D. Ting, Streamed approximate counting of distinct elements: Beating optimal batch methods, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014, pp. 442–451.

[21] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.

[22] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: International Colloquium on Automata, Languages, and Programming, Springer, 2002, pp. 693–703.

[23] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, Elsevier, 2002, pp. 346–357.

[24] R. Ben-Basat, G. Einziger, R. Friedman, Y. Kassner, Heavy hitters in streams and sliding windows, in: IEEE INFOCOM 2016-the 35th Annual IEEE International Conference on Computer Communications, IEEE, 2016, pp. 1–9.

[25] Q. Xiao, S. Chen, Y. Zhou, M. Chen, J. Luo, T. Li, Y. Ling, Cardinality estimation for elephant flows: A compact solution based on virtual register sharing, IEEE/ACM Trans. Netw. 25 (6) (2017) 3738–3752.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.

[27] D.E. Taylor, J.S. Turner, Classbench: A packet classification benchmark, IEEE/ACM Trans. Netw. 15 (3) (2007) 499–511.

[28] G.K. Zipf, Selected Studies of the Principle of Relative Frequency in Language, Harvard university press, 1932.

[29] P. Numpy, Zipf distribution generator, 2020, https://numpy.org/doc/stable/reference/random/generated/numpy.random.zipf.html.

[30] CAIDA, Center for applied internet data analysis, (2021). https://www.caida.org.

[31] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, G. Carle, Moongen: A scriptable high-speed packet generator, in: Proceedings of the 2015 Internet Measurement Conference, ACM, 2015, pp. 275–287.

[32] The Linux Foundation (FD.io project), Vector packet processor, white paper, Tech. rep., 2017, https://fd.io/wp-content/uploads/sites/34/2017/07/FDioVPPwhitepaperJuly2017.pdf.

[33] The Linux Foundation, Fd.io (fast data - input/output) project, 2020, https://fd.io.

[34] [Data plane development kit] (2021). https://dpdk.org.

[35] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi, High-speed software data plane via vectorized packet processing, IEEE Commun. Mag. 56 (12) (2018) 97–103.

[36] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, V. Sekar, Nitrosketch: Robust and general sketch-based monitoring in software switches, in: Proceedings of the ACM SIGCOMM, 2019, pp. 334–350.

Valerio Bruschi is currently a Ph.D. student in Electronic Engineering at the University of Rome "Vergata", working in the Network Programmability Laboratory. From 2015 to 2017, He worked as a Researcher for CNIT, participating to two different EU projects on stateful network dataplanes: BEBA, SUPERFLUIDITY. Later, He spent one year as a 'R&D engineer' at Telecom ParisTech for a research collaboration with Cisco about Vector Packet Processor, focused on Access Control List. He completed the Master's Degree in 'ICT and Internet Engineering' at the University of Rome "Tor Vergata" in May 2018.

Salvatore Pontarelli is a CNIT Senior Researcher at University of Rome Tor Vergata. He received his M.Sc. in electronic engineering from University of Bologna in 2000 and his Ph.D. from the University of Rome Tor Vergata in 2003. He participated in several national and EU funded research programs. His research interests include hash based structures for network applications, HW design of SDN devices and stateful programmable data planes.

Jerome Tollet is Distinguished Engineer working in the Cisco Chief Technical and Architecture Office (CTAO) with a specific focus on Datacenter/Container Networking, Policy and Security. He is leading networking-vpp project. Before being at CTAO, Jerome was driving SDN & NFV initiatives for a company he founded 16 years ago, and was CTO of. He has been participating and contributing to the Open Networking Foundation (ONF), ETSI Industry Specification Group on NFV, IETF Service Function Chaining (SFC) Working Group and various open source initiatives including OpenDayLight, OpenStack and OpenVirtualSwitch.

Dave Barach is a Cisco Fellow specializing in high-speed packet processing on commodity hardware. Dave has worked on the fd.io vpp stack for more than a dozen years (and counting). Dave started programming in 1968, and has written all sorts of high performance software as well as highly scalable performance analysis tooling.

Giuseppe Bianchi is Full Professor of Telecommunications at the University of Roma Tor Vergata. His research activity, documented in about 170 peer-reviewed papers accounting for more than 16,000 citations, spans several areas including wireless LANs, privacy and security, design and performance evaluation of broadband networks, network monitoring. He is in the editorial board for IEEE/ACM transactions on Networking, IEEE Transactions of Wireless Communications, and Elsevier's Computer communication. He has (co-)chaired more than 10 international IEEE/ACM conferences/workshops, the most recents being IEEE WoWMoM 2010 and ACM WinTech 2011, and the next major one being IEEE Infocom 2014 as technical co-chair.