Designing Probabilistic Flow Counting over Sliding Windows

Designing Probabilistic Flow Counting over Sliding Windows

Alessandro Cornacchia*, Giuseppe Bianchi†, Andrea Bianco*, Paolo Giaccone*
* Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, 10129, Italy
†CNIT / Università degli Studi di Roma - Tor Vergata, Via del Politecnico 1, Roma, 00133, Italy

***Abstract*—Probabilistic approaches allow designing very efficient data structures and algorithms aimed at computing the number of flows within a given observation window. The practical applications are many, ranging from security to network monitoring and control.**

We focus our investigation on approaches tailored for sliding windows, that enable continous-time measurements independently from the observation window. In particular, we show how to extend standard approaches, such as Probabilistic Counting with Stochastic Averaging (PCSA), to count over an observation window. The main idea is to modify the data structure to store a compact representation of the timestamp in the registers and to modify coherently the related algorithms. We propose a timestamp-augmented version of PCSA, denoted as TS-PCSA, and compare it with state-of-the-art solutions based on HyperLogLog (HLL) counters that evaluate the cardinality over a sliding window, but without storing the timestamps. We will show that TS-PCSA with a limited memory footprint is achieving a different tradeoff between memory and accuracy with respect to HLL-based solutions.

I. Introduction

Monitoring the number of distinct flows that are active within a network traffic aggregate is a crucial task for network managers. Several applications, including intrusion detection [1], [2], [3], traffic engineering [4], packet scheduling and router design [5], can benefit from a fast and accurate estimation of such flow cardinality. Different applications adopt different *flow* definitions and look at different traffic aggregates. For example, in a DDoS attack, several sources flood a victim host with a huge amount of traffic in order to make it unavailable. Therefore, a DDoS detection system should count the number of distinct source IP addresses (i.e., flows) that are currently active within the portion of traffic destined to a single host. Similarly, detecting a port scanning attack can be performed by counting the distinct destination ports in a set of packets with the same source IP address.

The capability to track in continuous-time the traffic statistics of interest is an essential property for new generation measurement algorithms. Existing literature about *streaming algorithms* for cardinality estimation [6], [7], [8], [9] has vastly addressed how to get accurate count estimates by processing the input traffic stream using a constant-time per-packet operations and logarithmic (or sub-logarithmic) memory footprint compared to the input stream size. Unfortunately, most of these widely adopted count-unique sketch data structures (e.g., PCSA, HyperLogLog (HLL) [8]) do not provide natively the possibility to devise a *sliding window* approach in a way to forget outdated information and consider only "recent" traffic. Rather, the typical deployment assumption is to operate the sketch structure on slotted time intervals (i.e., *epochs*) and reset the entire structure at the end of every interval before starting the next one. By working with static and non-overlapping measurement intervals, this simple solution cannot answer queries about past windows continuously over time, but only synchronously to interval boundaries (i.e., reset times). Thus, it may fail to detect the spreading behavior of a traffic aggregate if it happens in the middle of two consecutive epochs, as the corresponding flow cardinality will be split across two independent estimates. Furthermore, it is challenging to set the proper interval size, as it introduces a subtle trade-off between latency and detection capability. A large interval would introduce high reaction delays (being results available only at the end of the interval), whereas short intervals might not be able to spot a slow spreader. In our work, we will instead consider native approaches to support continuous-time measurements.

Two types of solutions can be devised to estimate the cardinality in continuous time, which we refer to as *timestamp-augmented* and *timestamp-free*. Timestamp-augmented algorithms [5], [10], [11] enhance existing count-unique sketches by augmenting the information maintained in the sketch structure with timestamp data. In a different line of work, timestamp-free solutions do not require any timestamp to be stored. For example, ST-HLL [12] approximates a triangular-shaped low-pass filter by running a periodic *staggered* reset of HyperLogLog registers. Therefore, in contrast to timestamp-augmented solutions, ST-HLL comes at zero memory extra cost, but its output might differ from the exact value of cardinality on a sliding window, that instead corresponds to filtering with a rectangular impulse response.

In this paper, we focus on timestamp-augmented approaches and we consider in particular Probabilistic Counting with Stochastic Averaging (PCSA) [6]. We show how the extend it to support timestamp-augmented measurements and discuss how to properly dimension the number of bits for the timestamp representation.

In summary, our main contributions are the following:

- we propose TS-PCSA, a basic timestamp-augmented algorithm to enable PCSA counting distinct flows over sliding windows.
- we design an optimized version of the algorithm, that can reduce the overhead of storing high-resolution times-

tamps. We combine low-resolution timestamps with stochastic averaging among time-shifted registers to smooth the error introduced by rounding.
- we highlight the possibility of further reducing the memory footprint of TS-PCSA, by studying the refresh time of the data structure storing the timestamps.
- we compare the accuracy of TS-PCSA+ with HLL-based solutions, and show their different tradeoff between accuracy and memory footprint.

The rest of the paper is organized as follows. In Sec. II we provide the necessary background to probabilistic data structures for cardinality estimation. We discuss both cumulative counting algorithms and their extensions to deal with the need of tracking, in continuous-time, cardinalities over sliding windows. In Sec. III we present our algorithm TS-PCSA and discuss its time and space complexity, where the latter is strongly dependent on how the timestamps are represented. In Sec. IV we propose TS-PCSA+, an optimization of the basic algorithm, highlighting possible ways to further optimize its memory footprint. We assess the performance of TS-PCSA+ in Sec. V, where we compare the achieved tradeoff between memory and accuracy with respect to timestamp-augmented and timestamp-free algorithms. Finally, we draw our conclusions and suggest future directions in Sec. VI.

II. Background on probabilistic counting

A. Cumulative counting

We now recall the technique of two classical probabilistic data structures, i.e., HyperLogLog (HLL) [8] and Probabilistic Counting with Stochastic Averaging (PCSA) [6], devised for cardinality estimation. We refer to these baseline sketches as *cumulative counting* techniques, as they do not provide any means to forget outdated flows, but rather increase over time their current estimation with the contribution of new arrivals. The goal is to evaluate the cardinality n of a set of flows X contained in a given stream of packets, i.e., with $|X| = n$. Now, for each packet in the traffic stream, we can evaluate an hash function $h(x)$ on its flow identifier $x \in X$ and compute the position[1] $p(h(x))$ of the left-most bit equal to 1 in the binary representation of $h(x)$. Let $R(x) = p(h(x))$ be the *rank* of a flow x and observe that $\text{Prob}(R = r) = 2^{-r}$, i.e., R follows a standard geometric distribution, given the uniformity property of the hash function. Note that, by construction, packets belonging to the same flow have the same value of the hash function and thus multiple packets of the same flow are counted just once.

For the moment, assume to store, as follows, all the ranks in the same bitmask with entries $C[b]$, where b is the bth Least Significant (LS) bit. For each flow x, set $C[R(x)] = 1$ to store the corresponding rank. After inserting 2^k flows, *ideally* we would expect to observe C composed of a block of zeros in the Most Significant (MS) bits and a block of k ones in the LS bits. Thus, we would estimate the cardinality as 2^{k_1} where k_1 is the number of ones in C. Unfortunately, due to the randomness of the traffic and due to the hash function, C is typically composed of a block of zeros (MS bits), followed by non-continuous sequences of zeros and ones, and finally a block of only ones (LS bits). Thus, we need to approximate the cardinality with an estimator, computed according to different approaches:

- in PCSA, consider the entire bitmask of ranks and approximate the number of distinct flows as $n = 2^{k_1}$ where k_1 is the size of the rightmost one block, or equivalently $k_1 + 1$ corresponds to the position of the first zero in C starting from the least significant bit;
- in HLL, consider only the maximum rank in the bitmask, i.e., $R_{\max} = \max_{x \in X} p(h(x))$, and approximate the distinct number of flows as $n = 2^{R_{\max}}$.

It is worth noticing that PCSA requires the entire bitmask C to be stored in a memory *register*, whereas HLL only the value of the maximum rank. Thus, HLL can be deployed using smaller memory registers than PCSA, by a logarithmic factor in the number of bits.

The above techniques are characterized by a large variance, due to the possibility of outlier flows (i.e., flows with $R \gg \log_2 n$) that would blow up their estimation. HLL and PCSA reduce such variance through *stochastic averaging*, introduced in [6]. Stochastic averaging splits uniformly at random the traffic stream into m traffic substreams, each substream updating a different memory register, according to the logic described above. Therefore different registers lead to m independent estimators, which are then averaged to reduce noisy fluctuations.

B. Counting over sliding windows

Timestamp-augmented sketches. Timestamp-augmented solutions [5], [11], [13] are characterized by augmenting the synoptic information contained in a probabilistic counting sketch with a time tag. Since the sketched information is temporally tagged, these techniques can distinguish and disregard outdated information with arbitrary precision by choosing a sufficiently high timestamp resolution. In fact, by properly choosing the resolution, they guarantee to use only the knowledge coming from traffic within the window. As a downside, managing timestamps poses a few challenges for implementation on resource-constrained programmable switches [14], due to their storage cost and the complexity of ignoring outdated entries at query time.

The sliding HyperLogLog algorithm [11] (W-HLL) is a state-of-the-art sketch for cardinality estimation based on timestamp values. The core idea is that if the exhaustive storage of all ranks observed in a past window would be memory sustainable, then a sliding window solution could be achieved by trivially re-running a vanilla HLL only on this rank subset upon each query. As a main contribution, W-HLL maintains only those ranks eligible to become maxima in the future as follows. It uses m Lists of Possible Future Maxima (LPFM) — in place of single-value registers — containing pairs $\langle$*timestamp, rank*$\rangle$. When a new rank R is inserted in a list, W-HLL evicts (1) all ranks $R' \leq R$ and

[1]Counted starting from one.

1: **procedure** QUERY (t,W)
2: $a = 0$ ▷ Init the accumulator for the average
3: **for** $i \leftarrow 0 \rightarrow (m-1)$ **do** ▷ For each counter
4: $\delta_i = \frac{\tau}{m-1} i - \frac{\tau}{2}$ ▷ Compute the temporal offset
5: **for** $k \leftarrow 0 \rightarrow (K-1)$ **do** ▷ For each bit starting from LS bit
6: **if** $T_i[k] + \delta_i < t - W$ **then** ▷ Check timestamp
7: **break** ▷ Leave the search loop if outside the window
8: $a = a + k$ ▷ Accumulate k for the average
9: $n = m \times 2^{a/m}$ ▷ Compute the average and the final count
10: **return** $n/0.775/W$ ▷ Output the final rate with bias compensation

Fig. 1: Querying at time t a TS-PCSA sketch (● code) or a TS-PCSA+ sketch (●● code), tracking a sliding window W.

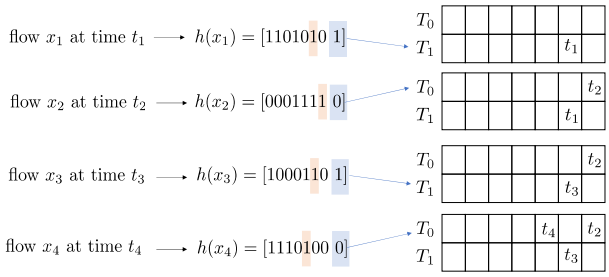

Fig. 2: TS-PCSA example when inserting the first 4 flows.

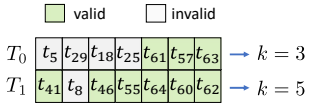

Fig. 3: The TS-PCSA sketch after inserting all 64 flows.

(2) all outdated ranks whose timestamp is oldest than a time window. Therefore, the arrival of a large rank evicts several smallest LPFM entries. W-HLL is functionally equivalent to to the exhaustive storage solution, but with significantly lower memory consumption.

Timestamp-free sketches. A different family of algorithms aims at approximating a sliding window filter without dealing with timestamps. As an example, the Staggered HyperLogLog [12] algorithm builds upon the idea of resetting one HLL register every τ unit of time, in a circular fashion. This implies that the data structure stays in a "warm" state, as only one register at a time loses its statistic. When a query is issued to ST-HLL, its registers span time windows with different lengths and misaligned with each other. By proper compensation, this sketch can approximate continuous-time operations at the same computational complexity of vanilla HLL, making it attractive for data-plane deployment on resource-constrained programmable switches.

III. TS-PCSA ALGORITHM

In the TimeStamp-augmented version of PCSA, denoted as TS-PCSA, we assume to have m arrays (denoted as "registers" with abuse of language), each of them storing K timestamps. We select m being a power of 2. This structure mimics the standard PCSA with m registers and K bits for each register, but each bit of the PCSA register is instead storing a b-bit timestamp. Our algorithm does not restrict the capabilities of a standard PCSA and it supports the same operations. In Sec. III-A we overview the ADD() and QUERY() operations in TS-PCSA, discussing their complexity and the timestamp representation in Sec. III-B.

A. Algorithm overview

Adding a flow is substantially equivalent to a standard PCSA. The ADD() operation exploits the $\log_2 m$ LS bits to choose a register (i.e., select the substream), while the remaining bits are used to compute the rank of the flow. Differently from PCSA, TS-PCSA updates the register in the position identified by the rank with the flow arrival time, instead of with a single bit.

Querying the flow cardinality in the last W observation window at time t is supported through the QUERY() operation, whose pseudocode is reported in Fig. 1. We temporarily ignore the code sections referring to TS-PCSA+, that will be clarified later in Sec. IV-B. Let T_i be the ith register and $T_i[k]$ be the timestamp stored in the $(k+1)$th LS bit in PCSA, and denote as *valid* all the timestamps that fall within the observation window. The main idea is to find across all the registers the average position at which the timestamp becomes invalid, starting from the rightmost position. The main loop (ln. 3-8) finds for each register such position (equal to k in ln. 8) and computes the average a/m, from which the cardinality is derived $\propto 2^{a/m}$ (ln. 9). Finally, an estimate of the flow arrival *rate* is computed by applying the same bias correction of PCSA and dividing by the length of the observation window.

Example. Fig. 2 shows a toy example of a TS-PCSA with 2 registers, each storing 7 timestamps. The traffic is constituted by a sequence of 64 flow arrivals, with flow x_i arriving at time t_i. For ease of explanation, we assume flows consisting of a single packet. By applying the hash function $h(x_i)$, the last bit is used to select the register, whereas the first 1 in the remaining binary string, starting from the LS bit, i.e. rank, identifies the position in the register where the timestamp is updated. The figure shows the step-by-step state when the first 4 flows are added with ranks $(2, 1, 2, 3)$. The final state at time t_{64}, after having inserted all flows, is shown in Fig. 3, where we highlight that most of the initial timestamps have been overwritten by the most recent ones, especially in low-rank positions which are more likely to be updated. Assume at this time to query the sketch using an observation window of length $W = t_{64} - t_{32}$. For register T_0, the first invalid timestamp is found at the 4th register entry ($k = 3$), since $t_{25} < t_{64} - W$, while for register T_1 in the 6th entry ($k = 5$), since $t_8 < t_{64} - W$. Thus, the average number of continuous blocks of valid timestamps is $n = (3 + 5)/2 = 4$, that is used to estimate the total number of flows as $2 \times 2^4 = 32$ (by chance, corresponding to the exact value of flows observed in W). For simplicity, in this example, we have not considered the bias correction factor.

B. Algorithm overhead and complexity

Memory consumption. The memory footprint of a TS-PCSA sketch using m arrays of b-bits timestamp and capable of counting up to 2^K unique flows is $m \times K \times b$ bits. Dimensioning b for a binary representation of the timestamp is not trivial. Notably, representing time with infinite precision would require an infinite number of bits. Thus, it is necessary to set a *time resolution*, defined as τ. Now the observation window W can be seen as divided into W/τ timeslots, for which at least $\lceil \log_2(W/\tau) \rceil$ bits[2] are required. The timestamps will be wrapped to the maximum integer representation chosen for the timestamp, and we need to be sure to properly compute differences between timeslots. Assuming to prune all the invalid timestamps periodically once every αW time (i.e., all timestamps before $t - W$ are reset), with $\alpha > 0$, we need to cover an interval of time $(1 + \alpha)W$ with distinct timeslots to properly compute the difference of time. We need also an additional bit to tag a timestamp as invalid. Thus, the total number of bits is $b = 1 + \lceil \log_2((1 + \alpha)W/\tau) \rceil$.

Time complexity. As regards the ADD() operation, TS-PCSA preserves the same $O(1)$ average time complexity of PCSA. The QUERY() operation is a bit more involved. TS-PCSA requires finding the first invalid timestamp in all registers to average their positions. Thus, querying the sketch has complexity $O(mK)$ due to a linear search in each register. We observe that in PCSA the linear search can be avoided with simple workarounds, like keeping a pointer to the first invalid position within each register. This is practicable because blocks of contiguous 1s cannot fragment once they have built up. In our algorithm the timestamps may become invalid after W time units have elapsed since when they were stored. Thus, a block of contiguous valid timestamps will likely fragment, hindering the use of such a simple technique. This substantial difference represents a limitation of our approach.

IV. Algorithm optimizations in TS-PCSA+

A. Timestamp rounding errors

When flow x_i arrives, it is associated with an integer timestamp t_i, which is a multiple of τ. Different ways can be used to round the actual arrival time to the slotted time. We will see later that rounding it to the closest timeslot is the strategy that minimizes the counting error, as could be expected intuitively. Nevertheless, the cardinality estimation TS-PCSA still suffers from some temporal rounding errors, highlighted in Fig. 4. Indeed, when a flow arrives after the middle of the timeslot, as in the case of x_1, the cardinality in W is over-estimated for $< \tau/2$ units of time. On the contrary, when it arrives before the middle of the timeslot, as in the case of x_2, the cardinality in W is under-estimated for $< \tau/2$ units of time. The overall effect, as shown later in our experimental analysis, is that the cardinality estimation appears as a *sawtooth* function around the average.

[2]We denoted by $\lceil\ \rceil$ the operation of ceil integer rounding

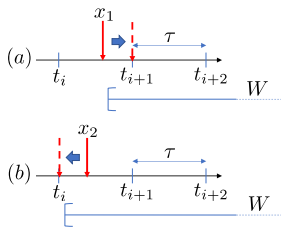

Fig. 4: Effect of rounding the arrival time on the cardinality estimation: (a) overestimation case, (b) underestimation case.

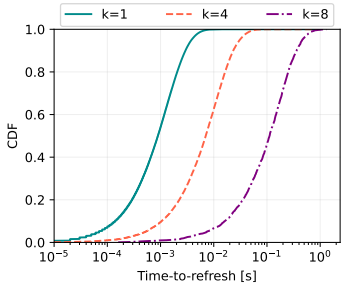

Fig. 5: Distribution of the timestamp refresh time for three different positions within a register.

B. Low-bit time quantization with register offsetting

To compensate for the systematic estimation errors due to the timestamp rounding, we propose an enhanced version of the algorithm, denoted as TS-PCSA+. It can be implemented by adding a single line of code to basic TS-PCSA (Fig. 1). As reported in the pseudocode, TS-PCSA+ introduces an offset δ_i for each register T_i, computed such that $\delta_0 = -\tau/2$ and $\delta_{m-1} = \tau/2$. Remember that in the PCSA family, individual registers can be seen as independent estimators, each giving contribution $2^{k/m}$ to the final count (see ln. 9). From the discussion in Sec. IV-A, it's easy to see that all estimators $2^{k/m}$ follow a sawtooth pattern over time. Now, the rationale is that by anticipating half of the sawtooth-like estimators and delaying a half — with the average phase offset being null — the phases combine destructively, averaging out the rounding error. We prove the effectiveness of this approach in Sec. V-B, which allows to significantly reduce the timestamp size, while preserving accuracy.

C. Further timestamp optimizations

The amount of bits for each timestamp has been assumed to be constant across all the positions within each register. We wish now to highlight that this is a suboptimal design choice since it is possible to reduce the number of bits, depending on the position within the register.

Indeed, consider a toy scenario, with periodic flow arrivals at rate $R = 10^5$ flows/s, and TS-PCSA+ to update $m = 64$ registers. Fig. 5 shows the CDF of the refresh time for each position of a register. It can be easily shown that, on average, the timestamp in position k will be refreshed every $(1/R) \times 2^k \times m$, which is coherent with the median value observed in the figure. As an extreme case, looking at the

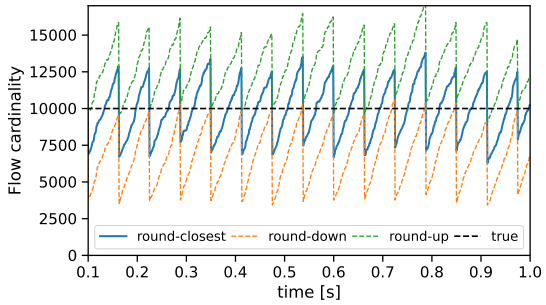

Fig. 6: Rounding effect in TS-PCSA with $b = 5$ bits.

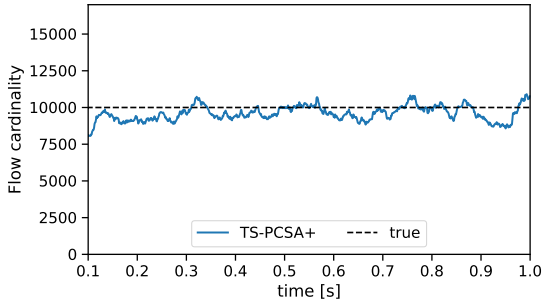

Fig. 7: TS-PCSA+ algorithm smoothes the sawtooth behavior.

graph the first position (i.e., $k = 1$) is almost surely updated within 0.01 s, suggesting that, by considering any window W larger than this value, storing the timestamp is useless. This suggests that it is possible to reduce the memory footage by never storing the timestamps in such a position. In general, by observing the CDF it is clear that some lower positions within the register can be omitted. At the same time, consider that TS-PCSA+, by construction, does not consider the timestamps within a register above an invalid timestamp (e.g., consider t_8 and t_{20} in Fig. 3), thus suggesting that also keeping the full bit representation for such position is useless. In summary, only the timestamp within a "reasonable" central range of positions should be stored to minimize the memory footprint of TS-PCSA+. Furthermore, different time resolutions could be considered depending on the position within the register. We leave these research directions for future work.

V. Performance Evaluation

We conducted a set of experiments on synthetic and real-world traffic workloads to evaluate TS-PCSA. In this section, we show (1) the effectiveness of our technique in compensating the overestimation and underestimation errors introduced by low-resolution timestamps and (2) we compare TS-PCSA to related continuous-time probabilistic counting solutions, including Sliding HyperLogLog [11] and Staggered HyperLogLog [12]. Results show that our algorithm improves accuracy upon existing timestamp-based methods, especially when few registers are available.

A. Experimental setup

Implementation. We developed all algorithms as Python routines. The source code is released under an open-source license and published online[3]. In our implementation, all algorithms use the SHA1 hash function. We extract from the SHA1 digest the first 32 bits, which are enough to rule out the problem of hash collisions for all the configurations of workload and window size we tested.

Workloads. We use two kinds of network traffic workloads.

1) *Synthetic*. We create new packet arrivals according to a periodic generation process with a deterministic rate λ. In this dataset the traffic stream is composed only of packets belonging to distinct flows, therefore each flow consists of a single packet. This baseline scenario is helpful in Sec. V-B to show the effect on cardinality estimation of rounding packet arrival times to coarse-grained time bins.
2) *CAIDA equinix-nyc*. A network traffic trace collected from a 10 Gbps link in an Internet backbone router from CAIDA [15]. We use the 2-tuple of source IP address and destination IP address as flow key. The trace refers to about 1 minute of traffic and contains about 2 million flows and 36 million packets.

Unless otherwise stated, the observation window W was set to 100 ms for all experiments.

Hardware setup. We run all experiments on Linux machines in an HPC[4] cluster. Each machine is equipped with two 2.10 GHz Intel Xeon Scalable Processor Gold 6130 CPUs with 16 cores and 384GB DDR4ECC RAM.

B. Effectiveness of our TS-PCSA+ optimizations

First, we provide evidence about how our technique based on register offsets averages out overestimation and underestimation errors introduced by time quantization. We measure flow cardinality on the synthetic workload using the baseline version of our algorithm (without register offsets). In Fig. 6 we compare round-up, round-down, and round-closest rounding strategies, where packet arrival times are set to the past, next, and closest represented timestamp, respectively. In this experiment, we used a TS-PCSA sketch with 256 registers, the time resolution τ was set to 0.0625 s. In all scenarios, we observe a sawtooth pattern in the flow cardinality estimate, with 16 peaks in a 1 s interval (the initial transient has been removed), coherently with what was discussed in Sec. IV-A. Adopting the round-up strategy, we only suffer overestimation errors. In fact, the minima correspond almost exactly to the true value. The contrary holds for round-up. Fig. 7 shows what happens when the same workload undergoes TS-PCSA+, our enhanced version. A relative time offset between registers of about $\tau/m \simeq 0.24$ ms almost cancels the peaks in the estimation.

C. Performance over real traffic traces

As a subsequent step, we analyze how our algorithm behaves under the realistic traffic workload described in Sec. V-A. As a performance metric, we measure the average

[3]GitHub repository: https://github.com/alessandrocornacchia/Stag-HLL.git
[4]Academic Computing Center at Politecnico di Torino - http://hpc.polito.it

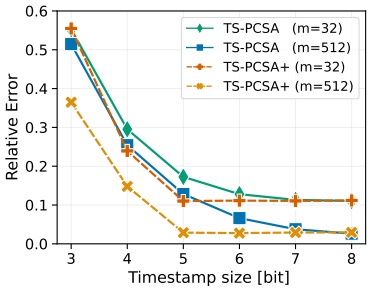

Fig. 8: Trade-off between accuracy and number of bits used for time quantization for TS-PCSA and TS-PCSA+.

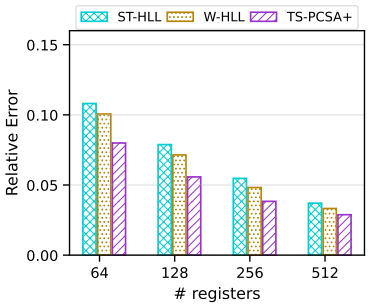

Fig. 9: Comparison between timestamp-augmented and timestamp-free algorithms over Internet traffic traces.

estimation error, relative to the true value of the cardinality.

How many bits can TS-PCSA+ save? We first try to understand how much we can gain in practical scenarios comparing TS-PCSA with its optimized version TS-PCSA+. We want to quantify the gain in terms of how many bits per timestamp we can save, without sacrificing accuracy. Fig. 8 shows that for all configurations in the number of registers, TS-PCSA+ requires as much as 38% less memory with respect to TS-PCSA to achieve a relative error below 10%. Performance stabilize at 5 bits/timestamp. We verified that even if timestamps were represented using python's *float32*, the relative error converges close to the same value. This means that our simple yet powerful offset technique closely approaches a system with "ideal" timestamp resolution.

How does TS-PCSA+ compare with state-of-the-art? We test TS-PCSA+ against W-HLL and ST-HLL that were introduced in Sec. II-B). Since all these sketches share the same structure based on registers, we configure them with an equal number of registers for comparison. TS-PCSA+ outperforms both timestamp-augmented and timestamp-free solutions (Fig. 9), especially in configurations with few registers. The reason is that W-HLL and ST-HLL base their estimations only on the most recent maximum ranks, whereas TS-PCSA+ considers a contiguous block of ranks not older than W, which is a piece of richer information. Notably, TS-PCSA+ can be deployed at a memory cost comparable to W-HLL. The space needed by W-HLL is a function of the LFPMs size, but bounded by $40m\ln(n/m)$ bits, being n the flow cardinality within the window. In its 5 bits/timestamp configuration, the TS-PCSA+ sketch is more lightweight than W-HLL whenever $me^{K/8} \leq n$ is satisfied. Considering that n is ranges between 10-20k for $W = 0.1$s, this is easy to be enforced with $K \leq 27$. ST-HLL gives the worst performance but is at least $8\times$ more lightweight in terms of memory footprint [12].

VI. Conclusion and Future Work

In this paper, we addressed the challenging task of counting unique flows over a sliding window in continuous-time. We proposed a novel timestamp-augmented sketch based on the PCSA probabilistic data structure and tested it over real-world Internet traffic. Thanks to the simple yet effective strategy to associate a constant temporal offset to the sketch registers, our algorithm remains as lightweight as previous techniques based on timestamp, however, it is up to 35% more accurate. We highlighted the opportunity for further compression of the TS-PCSA structure, which deserves a deeper investigation that we leave as future work. Finally, we think that a valuable research direction is to investigate how to efficiently exclude outdated timestamps at query time, which remains a shortcoming in our algorithm.

References

[1] V. Bruschi, S. Pontarelli, J. Tollet, D. Barach, and G. Bianchi, "Flowfight: High performance–low memory top-k spreader detection," *Computer Networks*, vol. 196, p. 108239, 2021.

[2] Y. Chabchoub, R. Chiky, and B. Dogan, "How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?" *EURASIP Journal on Information Security*, vol. 2014, no. 1.

[3] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers and Security*, vol. 28, no. 1, pp. 18–28, 2009.

[4] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 547–558, 2015.

[5] H.-A. Kim and D. O'Hallaron, "Counting network flows in real time," in *IEEE GLOBECOM*, 2003.

[6] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[7] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms*. Springer, 2003, pp. 605–617.

[8] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA*, 2007.

[9] F. Giroire, "Order statistics and estimating cardinalities of massive data sets," *Discrete Applied Mathematics*, vol. 157, no. 2, p. 406–427, 2009.

[10] É. Fusy and F. Giroire, "Estimating the number of active flows in a data stream over a sliding window," in *ANALCO*, 2007.

[11] Y. Chabchoub and G. Heébrail, "Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window," in *International Conference on Data Mining Workshops*. IEEE, 2010.

[12] A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone, "Staggered hll: Near-continuous-time cardinality estimation with no overhead," *Computer Communications*, vol. 193, pp. 168–175, 2022.

[13] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, "Memento: Making sliding windows efficient for heavy hitters," in *CoNEXT*. ACM, 2018.

[14] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," in *ICNP*. IEEE, 2018.

[15] "CAIDA 2019," https://www.caida.org/catalog/datasets/trace_stats/nyc-a/2019/equinix-nyc.dira.20190117-130000.utc.df.txt, accessed: 05-2022.