

Received January 17, 2019, accepted February 10, 2019, date of publication February 13, 2019, date of current version March 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2899161

A Probabilistic Counting Framework for Distributed Measurements

NICOLA BONELLI¹, CHRISTIAN CALLEGARI^{1b2}, AND GREGORIO PROCISSI^{1b1}

¹Dipartimento di Ingegneria dell'Informazione, CNIT, Università di Pisa, 56122 Pisa, Italy

²Radar and Surveillance Systems National Laboratory of CNIT, Dipartimento di Ingegneria dell'Informazione, Università di Pisa, 56122 Pisa, Italy

Corresponding author: Christian Callegari (christian.callegari@cnit.it)

This work was supported in part by the University of Pisa under the PRA 2018-2019 Research Project "CONCEPT COmmunication and Networking for vehicular CybEr-Physical sysTems".

ABSTRACT The technological maturity attained by general purpose processors and network interface cards makes today's commodity PCs viable and high performing alternatives to specialized hardware for deploying network devices, such as switches, routers, and generic middleboxes. In addition, the flexibility of the software solution seems to be perfectly in line with the emerging trend towards the data-plane programming abstractions brought by recent proposals such as Openflow and the P4 language. However, if programming abstractions provide the way elementary instructions (primitives) are combined together, the development of such processing primitives is left to the network programmer. Although the type of such functions is strongly domain specific, we can safely assume that the *counting* primitive is easily required in a great deal of practical contexts. This paper presents a counting framework based on probabilistic sketches and *LogLog* counters for estimating the cardinality of large multi-sets of data. The proposed data structure is designed to be fast and compact for ready use in the on-line chain of processing of network devices running at multi-gigabit speeds. The complete implementation is provided within the *probabilistic data structures (pds)* library which has been designed, developed, experimentally assessed, and released as open-source for free download. Although the paper specifically presents two possible use-cases, the *pds* library can be used in rather general scenarios, even outside the networking domain.

INDEX TERMS Approximate counting, loglog counter, multi-set cardinality, network applications, reversible sketches.

I. INTRODUCTION

Traffic measurements and monitoring are routinely performed by network operators to assess the operational status of their infrastructure as well as to detect possible misbehavior and malicious activity. No matter of the specific goal, in the vast majority of the cases such investigations ultimately involve the primitive of *counting* packets or flows of packets having predefined characteristics. The application scopes can be manifold, from troubleshooting and sanity check, to traffic profiling for commercial use, or anomaly detection for network security. In all cases, however, the process of retrieving statistics from real measurements is dramatically complicated by the ever-increasing link bandwidth that nowadays requires measurements to be taken on-the-fly on top of at least 10 GB+ network segments.

The associate editor coordinating the review of this manuscript and approving it for publication was Ruilong Deng.

In such a network scenario, even an intuitively *easy* operation such as counting may become hard, as the time budget available i) to discriminate the data of interest first and ii) to select and increment memory registers easily drops below a few nanoseconds. Besides, high-speed links naturally induce likely large numbers of packets to be filtered first and counted later. This operation requires larger data structures which, in turn, must be placed in larger (but slower) DRAM memories. As a result, consolidated techniques, like those based on traditional hash tables, become practically unfeasible in many cases.

The whole frame gets further complicated whenever monitoring and measurements are carried out in a distributed fashion, i.e., by placing probes at multiple vantage points. In fact, this is now standard practice for network operators whose objective is to come out with higher-level aggregate statistics from single measurement collections. Cumulative statistics are typically performed by *mediators* in charge of analyzing and profiling traffic and possibly generating alerts

and executing mitigation actions in case of detection of suspicious anomalies. At this stage, a big issue to be addressed is to prevent mediators from accounting duplicated flows monitored at different probes. This problem may be solved either by computational intense post-processing operations on all single traffic dumps or (and this is the solution proposed in this paper) by adopting a streaming approach to elaborate data directly *on the wire* by means of suitable data structures that allow to automatically discard *by design* duplicated data when they get merged.

More generally, the huge volume of network traffic of today's Internet¹ pushes *scalability* as a primary issue and makes traditional monitoring approaches based on capture, storage, and post processing of data practically unfeasible and obsolete. As a result, a new trend of research in which a first, possibly coarse-grained, stage of processing is performed in the data-plane of the network nodes themselves while delegating more sophisticated analysis to a second tier of specialized systems (e.g., middleboxes, SIEMs, DPI modules) is rapidly emerging. This approach grounds back on the seminal paper [2] and has evolved quite a lot over the years and even strengthened by the SDN paradigm as the newly introduced flexibility may in fact readily enable the integration of monitoring capability into programmable nodes that implement processing *primitives* that can be instantiated through the control plane as real programs [3]–[6].

The goal of this research is to propose a general purpose and flexible counting framework that addresses at the *data-plane* level the previously discussed requirements, and that could be used as a *primitive* in a broad plethora of network applications. The adopted approach is that of “trading certainty for time/space” [7] by using probabilistic algorithms at the cost of a small and tolerable error rate. In a nutshell, the proposed counting framework combines an efficient probabilistic counter (e.g., the *LogLog counter* [8]) to compact probabilistic data structures (*sketches*), acting as containers, and that can be “reversed” only upon specific conditions are met. Overall, the whole data structure inherits the low complexity and memory efficiency of probabilistic counters and Bloom filter like structures, while proving insensitivity to data duplication and allowing the identification of the source of investigated data.

Beside the architectural view of the counting framework, the paper also presents a practical and efficient implementation of the proposed data structure. The implementation is made available to the community for free download.²

As a final remark, it is worth recalling here that the use of compact and reasonably “anonymous” data structures that keep aggregate statistics while still being able to show more specific details on single flows upon request and proper algorithmic reversal does also meet another big issue of network monitoring practice which is the compliance with current

legislation regarding *privacy preservation*. In particular, due to current legislation trends (see [9] for a detailed summary of the constraints imposed, for example, by the EU legislation), much of the information retrieved from the captured traffic is considered privacy-sensitive, and its disclosure and storage is subject to strict rules. Such constraints not only apply to verbose packet traces but a huge number of derived metadata (including per-flow counting reports) are considered to contain privacy-sensitive information and therefore their export and storage (if not wholly forbidden) is subject to stringent rules. As such, specific details on single flows should be disclosed only if strictly necessary (*necessity principle*) and with a level of granularity that must not be excessive concerning the purposes for which data are collected and further processed (*proportionality principle*).

This work unifies the previous research presented in [10] and [11] and extends their finding by refining the overall architecture, by integrating a more efficient counter, and by proposing a real high-performing C++ implementation of the whole counting data-structure.

The remainder of the paper is organized as follows. Section II provides a summary of the relevant related work and clarifies the contribution of the paper. Section III specifically provides two distinct motivating use-cases while Section IV gives the necessary theoretical background. The proposed counting framework is presented in Section V together with the specific declination in the previously presented use-cases. Section VI presents the implementation of the counting framework whose performance is assessed through the experimental results included in Section VII. Finally, concluding remarks end the paper.

II. RELATED WORK

Today's Internet can be seen as a complex and heterogeneous system that evolves at a ever growing speed. Its transformation process involves: the growth of the traffic volume [1], the proliferation of new services, the deployment of novel communication technologies and paradigms, and (last but not least) the increase of sophisticated cyberattacks against normal network operations. As such, network nodes have been increasingly called upon a much larger gamma of processing operations with respect to plain forwarding services. In this scenario, the quest for *programmable network nodes* that efficiently perform non trivial operations on the data plane directly is becoming more and more crucial in the design of a new generation of Internet systems [12].

Network measurements and monitoring do not make an exception to this trend and, in fact, network nodes themselves are increasingly becoming the first monitoring players. This trend is strongly fueled by the emerging approach towards an improved programmability of the data plane, beyond the elementary match/action abstraction provided by OpenFlow. In particular, a handful of proposals such as OpenState [3], FAST [13] and more recently [4] explicitly target stateful operations. The P4 programming language [5] further pushes the concept of data plane programmability by allowing the

¹According to Cisco [1], traffic volume will be soon measured in Zettabytes (1 billion terabytes), with more than 20 billions connected devices forecasted by 2020.

²<https://github.com/awgn/pds>

nodes to take local decisions, thanks to the availability of internal states in persistent memories. As a result, programs can run directly on the switch data path with maximum performance. For general purpose platforms, the PFQ I/O framework [6] provides a programming language especially devoted for packet processing [14], [15] in a streaming fashion based on a functional algebra [16].

In the monitoring and troubleshooting area, the above philosophy allows the data plane itself to collect in place and report network details without the intervention of the control plane. In this context, two examples of applications are the P4-based In-Band Network Telemetry (INT) [17] and the ETSI driven In Situ OAM (iOAM) [18].

However, programmability encompasses two complementary aspects: the elementary “instructions” (primitives) to be used by a program, and the way such instructions are combined together to permit the programmer to formally describe a desired behavior. While the above abstractions, such as P4, XFSM machines, etc., target the second aspect, the actual primitives are typically domain-specific and involve the use of algorithms and data structures that must be available at the node level. In this paper we specifically focus on the counting primitive which has been addressed in the literature by a great deal of scientific researches.

Opensketch [19] provides a quite general *software defined* framework for traffic measurement operations for commodity switches. More in detail, it provides the measurement APIs as well as a library of data structures (mainly, sketches) and primitives that run on the switch data plane upon a three-stage pipeline performing hashing, classification and counting, respectively.

The use of sketches (and, more generally, of randomized data structures) has been proposed many times in the literature to solve specific issues. Alon *et al.* [20] carried out a theoretical analysis and derived space efficient randomized algorithms for approximating the first three frequency moments of a sequence. Tristan *et al.* [21] recently proved the benefits of replacing standard counters with approximate counters in a machine learning algorithm running on a GPU. Also, they later provided a general framework [22] for adding approximate counters based on multiple algorithms. Estan and Varghese [2] proposed two novel and scalable algorithms based on counting Bloom filters for identifying the large flows based. The same authors later presented the use of virtual and multi-resolution bitmaps for Counting Active Flows [23]. Bandi *et al.* [24] address the frequent elements problem in networking system and propose TCAM based space saving algorithms for Network Processors (NPU). Cormode *et al.* used the sketch data structure to detect heavy hitters [25], [26] while the same data structure has been used within data streaming algorithms for flow size estimation [27]. Sketch based data structures have also been also used to derive accurate delay figures [28], to measure icebergs³ from distributed streams [29] and to design a space

and time efficient hybrid SRAM/DRAM counter architecture [30].

In 2007, Stanojevic [31] proposed the integration of probabilistic counters into sketches through the SAC and HAC algorithms. The objective was to reduce the memory footprint of counters so as to totally (SAC) or partially (HAC) meet the small size of fast SRAMs while keeping a bounded approximation error. With the same purpose, Hu *et al.* [32] proposed DISCO (DIScount COUNTing), a memory parsimonious algorithm for estimating flow size and volume. The DISCO algorithm consists of two phases for counting update and estimation inversion and the resulting data structure proves to entirely fit the SRAM of an Intel IXP2850 network processor.

CEDAR [33] makes a step ahead by decoupling the counter estimators from their estimation values. This allows to scale to arbitrary counters values and to achieve optimal min-max estimation error with same memory footprint of DISCO and SAC.

By separating flows into different buckets with different scales, the Independent Counter Estimation Buckets (ICE-BUCKETS) [34] algorithm allows to further improve the upper bound on the relative counting error of up to 57 times over real traffic traces.

The assistance of on-chip cache memory to process the most frequent elephant flows is instead advocated by CASE [35]. This allows to significantly increase the throughput of the counting framework while keeping bounded approximation error.

In the last year, two papers addressed the detection of heavy flows by means of *packet sampling* and may somewhat be regarded as competitors of our proposal. First, Afek *et al.* [36] elaborated a detecting algorithm for the SDN context under the match and action paradigm. Their proposal is based on the collaboration of OpenFlow Switches and Controller through the Sample&Pick technique. Still being an efficient method for detecting heavy hitters in the SDN context, differently from our approach, it involves the forwarding of the candidate packets to the controller and cannot be extended to the more general case of counting the number of packets of each flow, not being scalable enough (it would require to forward too many packets to the controller). Moreover, the proposed approach also relies on the possibility for the probes to mark the sampled packets (so as to avoid counting duplicates), which makes the method only applicable in specific contexts (e.g., VLAN).

Sampling is also at the core of the distributed counting algorithm proposed by Basat *et al.* [37]. This method surely represents the main competitor of our proposal, being able to solve the flow frequency estimation problem. Nonetheless, the two methods differ for several characteristics, among which the use of sampling, instead of sketches as in our algorithm, represent the main difference. This allows the method [37] to achieve a better memory footprint (roughly $O(\sigma^{-2})$, where σ is the target error parameter, instead of $O(\sigma^{-3})$), allowing to store a single counter instead

³Icebergs are defined as network traffic flows that have high aggregate volume across many different monitors

of a counter for each sketch bucket. Nonetheless, this comes at the expense of a bigger computational complexity, requiring each probe to maintain an ordered list of χ packets (with complexity $O(\log \chi)$, instead of $O(1)$ – complexity of the sketch) and the controller to merge the N lists sent by the N probes (with complexity $O(\chi \log \chi)$, instead of $O(1)$ as in our case). Moreover, it is worth noting that, differently from our method, the sampling phase in [37] implies that the algorithm could be unable to observe every flow (i.e., the case in which a given flow is not part of the samples), while our approach is able to see all of the flows. Although from a statistical point of view, an unobserved flow corresponding to a zero frequency estimation (due to sampling) is equivalent to a wrong estimation (due to counter errors), for some applications not being able to observe a given flow may have a different impact on the system performance (e.g., lawful traffic interception). Finally, but not less significant, our proposal is the only one that may comply with the current legislation on *privacy preservation*. As it will be elaborated in Section V-B, since the controller does not know the hashing scheme, it cannot reverse the sketches, hence cannot access any private information. Indeed, only the probes can reverse the sketches and only the one that created a specific sketch can identify the contributing flows, being the only one with the corresponding footprint Bloom Filter.

Our Contribution: The paper proposes a novel counting framework based on probabilistic sketches and LogLog counters for estimating the cardinality of large multi-sets of data. Apart from the already discussed peculiarities of our solution, it is also worth noting that to the best of our knowledge this paper is the first to also provide a software implementation of the proposed framework.

Indeed, all of the above works are either theoretical (not providing implementation), or propose device-specific *hardware* implementations.

However, the maturity attained by *Software Defined* approaches, the computational power of today's CPUs and network cards, and the availability of accelerated capture frameworks [6], [38]–[40] motivate our work to target general purpose platforms.

To this aim, we propose (and make available for download) the C++ *probabilistic data structures (pds)* software library implementing the data structures thereafter introduced. The library is built upon template meta-programming, thus providing robustness, composability, and ease of re-usability as all data structures are abstract containers of generic data type to be specialized upon need. As a result, the use of the *pds* library is not necessarily bound to network applications and can be conveniently adopted even in completely different contexts.

III. TWO MOTIVATING USE-CASES

This section presents two real-world use-cases in which network monitoring and security applications, respectively, require the use of efficient data structures for high-speed data processing. As it will be elaborated upon, the first

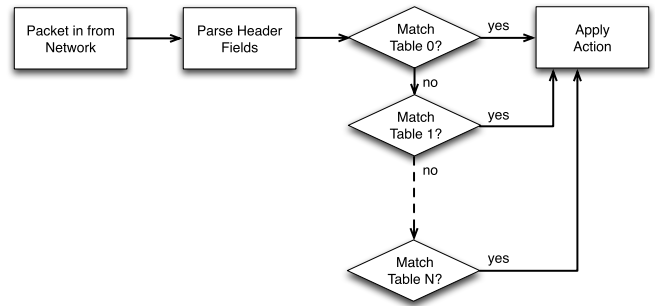


FIGURE 1. Packet processing in the switch.

use-case is induced by the recently emerged Software Defined Networking (SDN) paradigm and the new way of thinking of network applications according to this philosophy. The second use-case, instead, refers to a more traditional architecture of anomaly detection through distributed measurement points over backbone networks. In both cases, it will be evident that performance is not the only addressed requirement as the management of data collected from multiple vantage points and privacy preservation are not less important issues to handle.

A. SDN AND INTERNET EXCHANGE POINTS MONITORING

In a nutshell, from the functional point of view, an Openflow-based software defined network works according to the following paradigm. A switch, upon receiving a packet from the network (see also Fig. 1), first parses the packet headers to check for possible matches between these and the *fields* specified in any of the rules stored in the different flow tables. If a match is found, the corresponding *actions* (e.g., forward) are undertaken, otherwise, in the most common case, the packet is encapsulated into a new packet and sent to the controller through the secure channel. Hence, the controller will create a rule for such a packet and will update the switch flow tables accordingly.

It is important to highlight that the number of “unknown” packets (i.e., packets for which the switch has not a rule that specifies an action) should be low, during “normal” network behavior, for not overwhelming the controller and that, in any case, such packets can be considered as representative of some anomalous network behavior.

On the Internet, the different “independent” networks are connected through what is known as peer relationships. Hence, packets are forwarded towards the destination, passing from one network to another, because such networks have a peering relationship with each other. The point where the different networks interconnect and such peering relationships are established is named Internet Exchanges Point (IXP).

In a nutshell, at its most basic level, an IXP is a switch into which multiple networks connect and can then pass bandwidth. More realistically, an IXP is a Layer 2 network.

Given such a definition, it is clear that IXPs represent a “natural” deployment field for SDN technologies, (as an example, see SDX [41] and CARDIGAN Project [42]).

Such a scenario represents a good use-case for our proposed architecture. Indeed, it is well known that the security problems in an IXP [43] are usually due to:

- Unhygienic routers
- BGP manipulation
- Network capacity theft

While the first two issues can be solved (or, at least, significantly alleviated) by using a meticulous configuration of the IXP apparatus, the network capacity theft is mainly due to processing and forwarding of traffic flows that are not supposed to get through the IXP. Hence, it is clear that counting and identifying such “unknown” flows is of primary interest in such a context. In the following, we will show that our proposal proves to be well tailored to this case by providing the details of the algorithms as well as the results obtained from the experimental analysis.

B. ANOMALY DETECTION IN BACKBONE NETWORKS

In the last few years, many research groups have focused their attention on developing novel detection techniques, able to promptly reveal and identify network attacks, mainly detecting Heavy Changes (HCs) in the traffic volume [44]–[48]. Nevertheless, the recent spread of coordinated attacks, such as large-scale stealthy scans, worm outbreaks, and distributed denial-of-service (DDoS) attacks that occur in multiple networks simultaneously, makes the detection by using isolated intrusion detection systems that only monitor a limited portion of the Internet challenging. Hence, the research efforts are now moving to develop distributed approaches to solving such an issue [49].

In distributed anomaly detection algorithms, multiple detection probes – distributed in the backbone network – monitor a given portion of the network separately and report the collected information to a single location that analyzes the data and generates the alerts. By limiting the scope to the simplest case of anomaly detection algorithms that analyze traffic volumes only, the data collected by the probes are represented by the estimation of the number of traffic flows observed in a given time window. Hence, the first problem to be solved is to provide a reliable estimate of such quantities.

This task is performed by the distributed probes, that then forward the estimated data to the mediator that is responsible for aggregating them. It is worth noticing here that, when aggregating these structures at the mediator level, the problem of *not counting* duplicated flows – i.e., the flows observed by more than a single probe – must be solved.

In general, the counters values associated with specific flows represent sensitive users information and must not be openly disclosed. The use of probabilistic data structures like sketches [50] provides an ideal container to keep all such data in an aggregated and not directly accessible way. However, whenever traffic anomalies are detected, a method of

identifying the flows (in particular, IP addresses) responsible for that supposed misbehavior is needed. As it will be shown, the counting framework hereafter presented solves this issues by allowing a controlled reversal of suspicious data only, without no impact on regular (“normal”) traffic.

IV. THEORETICAL BACKGROUND

This section presents the minimal – but necessary – theoretical background on probabilistic counters and reversible sketches to allow the reader to understand the overall system architecture completely. Wherever needed, further details are left as references to relevant bibliographic items.

A. PROBABILISTIC COUNTERS: LOGLOG COUNTERS AND THEIR OPTIMIZATION

Probabilistic counters are intended as a class of algorithms that estimate the number of distinct elements in a set. Naturally, such an objective could be achieved exactly, at the expenses of a huge memory footprint when the number of elements to count becomes very large. In fact, this is precisely the case of traffic data on high-speed networks, especially in the presence of malicious packet flooding.

Probabilistic algorithms like LogLog counters [8], instead, “trade certainty for time/space” [7] by estimating the number of unique occurrences of elements in large multisets through impressively compact data structures at the expenses of a small error rate.

In this section, an optimized version of the LogLog counter will be used. Furthermore, the rationale of the algorithm, together with some of its improved variations, will be given.

More formally, given a multiset M produced starting from a discrete universe U , the objective is to estimate the cardinality of the support of M (aka its *dimension*), namely the number of *distinct* elements it comprises. Like in many similar algorithms, even in this case it can be assumed that a hash function $h : U \rightarrow \mathcal{U}$ is available for transforming each element of U into sufficiently long binary strings $x \in \mathcal{U}$ producing a “random” multiset $\mathcal{M} = h(M)$ with n distinct elements. Note that the use of hash functions allows obtaining strings x with random uniform independent bits.

Given that, let us suppose that the strings are infinitely long, that is $\mathcal{M} = \{0, 1\}^\infty$ (this is a convenient abstraction at this stage) and let $\rho(x)$ denote the (1 based) position of its first 1-bit. Consider now the set $\mathcal{P} = \{\rho_1, \rho_2, \dots, \rho_n\}$, with $\rho_i = \rho(x_i)$, $x_i \in \mathcal{M}$. The elements of \mathcal{P} form a sequence of independent and identically distributed geometric random variables with common pmf:

$$\Pr\{\rho = k\} = \left(\frac{1}{2}\right)^k, \quad k \geq 1 \quad (1)$$

Hence, the maximum of the set \mathcal{P} :

$$R(M) = \max_{x \in M} \rho(x) = \max_{1 \leq i \leq n} \rho_i \quad (2)$$

has the cumulative distribution function:

$$\Pr\{R \leq k\} = \prod_{i=1}^n \Pr\{\rho_i \leq k\} \quad (3)$$

$$= (\Pr\{\rho_i \leq k\})^n \quad (4)$$

$$= 1 - (\Pr\{\rho_i > k\})^n \quad (5)$$

$$= \left(1 - \left(\frac{1}{2}\right)^k\right)^n \quad (6)$$

and its mean value [51]:

$$\mathbb{E}[R] = \sum_{i=1}^n \Pr\{\rho_i > k\} \quad (7)$$

$$= \sum_{i=1}^n 1 - \left(1 - \left(\frac{1}{2}\right)^k\right)^n \quad (8)$$

$$\approx \log_2 n \quad (9)$$

provides a reasonably rough approximation of $\log_2 n$. In fact, it turns out [8] that the additive bias of R in estimating $\log_2 n$ is about 1.33, while its standard deviation is around 1.87.

To improve the estimate of n , the elements of \mathcal{M} can be divided into m groups (buckets) ($M^{(j)}$ with $j = 1, 2, \dots, m$) and compute the parameter R on the strings belonging to each bucket. Typically, $m = 2^k$ so that we can use the first k bits of x to represent the binary index of the bucket. For each group, let the registers $R^{(j)} = \max_{x \in M^{(j)}} \rho(\tilde{x})$, where \tilde{x} is obtained by discarding the first k bits of the strings $x \in M^{(j)}$ (indeed, the statistics on the position of the first bit set to 1 do not depend on the offset).

Then, the arithmetic mean:

$$\frac{1}{m} \sum_{j=1}^m R^{(j)} \quad (10)$$

is legitimately expected to approximate $\log_2(n/m)$, plus an additive bias.

Thus, the estimate of n according to the *LogLog* algorithm is:

$$E = \alpha_m m 2^{\frac{1}{m} \sum_{j=1}^m R^{(j)}} \quad (11)$$

where α_m is the bias correction factor in the asymptotic limit and can be evaluated as follows [8].

$$\alpha_m = \left(\Gamma(-1/m) \frac{2^{-1/m} - 1}{\log 2} \right)^{-m} \quad (12)$$

$$\Gamma(s) = \frac{1}{s} \int_0^\infty e^{-t} t^s dt \quad (13)$$

Notice that the algorithm needs to store m registers (the values $R^{(j)}$ computed over each buckets) each of them having potentially unlimited length. Durand and Flajolet [8] found that collecting only the $\theta_0 = \lceil 0.7m \rceil$ smallest values (*truncation rule*) and limiting their range to the interval $[0 \dots \lceil \log_2(\frac{n}{m}) + 3 \rceil]$ (thus limiting their memory occupancy to $\lceil \log_2 \lceil \log_2(\frac{n}{m}) + 3 \rceil \rceil$ bits) yields an estimation error

for the LogLog counter of $\frac{1.05}{\sqrt{m}}$. They name such an optimized version of this counter *Super-LogLog* counter.

A further decrease of the estimation error up to $1.04/\sqrt{m}$ has been achieved in [52] by introducing an other optimization named *Hyper-LogLog* counter. Roughly speaking, the performance improvement has been obtained by replacing the arithmetic mean of the LogLog counter with the geometric mean, namely:

$$E = \frac{\alpha_m m 2}{\sum_{j=1}^m 2^{-R^{(j)}}} \quad (14)$$

where α_m is given by:

$$\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+x}{1+x} \right) \right)^m dx \right)^{-1} \quad (15)$$

More precisely [52], if $\sigma \approx 1.04/\sqrt{m}$ corresponds to the standard error, the estimates provided by the Hyper-LogLog counter are expected to be within $\pm\sigma$, $\pm 2\sigma$, and $\pm 3\sigma$ of the exact count in 65%, 95%, and 99% of all cases, respectively.

As a final note, it is worth to elaborate upon the memory footprint of such probabilistic counters. Indeed, the main benefit of their use is represented by their extremely low memory occupancy which, in many cases, allows their placement into a reasonably small data structure to reside in small memories, such as fast caches or those on board of IoT devices. The overall memory occupancy of a counter depends on the cardinality of the support of the multiset to estimate as well as on the required estimation accuracy. Indeed, by construction, the memory footprint of any single LogLog counter is that of m registers (also called *small bytes*), each of them sized $\log_2 \log_2 N$ bits, where N is an a-priori estimate of the multiset dimension. Depending on the application, by adequately tuning the parameter m may provide a very compact counter and enable a significant computation speedup.

B. REVERSIBLE SKETCHES

Sketches have proven to be useful in many data stream computation applications [50], [53], [54]. Recent work on a variant of the sketch, namely the k -ary sketch, showed how to detect large changes in massive data streams with small memory consumption, constant update/query complexity, and provably accurate estimation guarantees [55].

In a nutshell, a sketch (see Fig. 2) is a two-dimensional $d \times w$ array $S[l][j]$, where each row l ($l = 1, \dots, d$) is associated with a given hash function h_l . These functions give an output in the interval $(1, \dots, w)$ and these outputs are associated with the columns of the array. As an example, the bucket $S[l][j]$ is associated with the output value j of the hash function l .

Considering the input data as a stream that arrives sequentially, item by item, where each item consists of a *hash key*, $i_t \in (1, \dots, N)$, and a *weight*, c_t , when new data arrive, the sketch is updated as follows:

$$S[l][h_l(i_t)] \leftarrow S[l][h_l(i_t)] + c_t \quad (16)$$

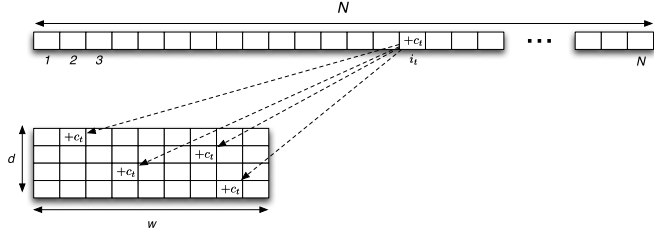
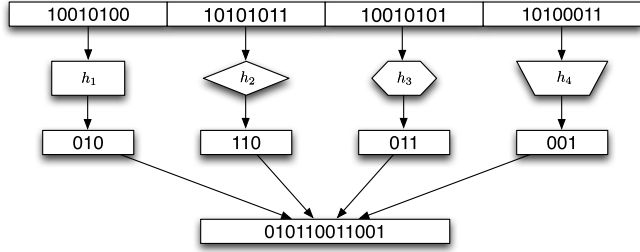
FIGURE 2. *k*-ary sketch.

FIGURE 3. Modular hashing.

The update procedure is realized for all the different hash functions as shown in figure 2.

However, sketch data structures have a major drawback: they are usually not reversible. In other words, a sketch cannot efficiently report the set of all keys that correspond to a given bucket.

Such limitation can be removed by modifying the input keys and hashing functions to make it possible to recover the keys with certain properties without sacrificing the detection accuracy. With this approach, the paper [56] proposes a novel algorithm for efficiently reversing sketches, by modifying the update procedure for the *k*-ary sketch through modular hashing and IP mangling techniques.

The modular hashing operates by first partitioning the n -bit long hash key x into q words x_1, x_2, \dots, x_q of equal length n/q such that $x = x_1|x_2| \dots |x_q$. Each word is then hashed separately by using a different hash function, h_{di} ($i = 1, \dots, q$), to obtain an m -bit long output. Finally, these outputs are concatenated to form the final hash value (see Fig. 3):

$$\delta_d(x) = h_{d1}(x_1)|h_{d2}(x_2)| \dots |h_{dq}(x_q) \quad (17)$$

Hence, the final hash value consists of $q \times m$ bits which, in turn, makes the number of column of the sketch equal to $w = 2^{q \times m}$.

Note that the use of the modular hashing may cause a highly skewed distribution of the hash outputs. Consider, as an example, the widely typical case in which IP addresses are used as hash keys. In network traffic streams there are strong spatial localities in the IP addresses since many IP addresses share the same prefix. As a result, the first octets (equal in most addresses) will be mapped to the same hash values increasing the collision probability of such addresses.

To efficiently resolve this problem, the *IP mangling* technique has to be applied before computing the hash functions. By using such a technique, the system randomizes, in a reversible way, the input data to remove the correlation or spatial locality.

Reversing Algorithm: The full description of the algorithm for reversing the sketch is given in [56]. In a few words, the rationale of the algorithm is to check separately all possible words of the keys to find a match with the corresponding portion of bucket address in the sketch. However, the use of hash modularity allows to immediately prune out of the whole cartesian products of words all of the q -uples in which even a single word does not hit any sub-address of the buckets to invert. This way the inversion algorithm converges very quickly and produce *all of the keys that may hit to the selected buckets* in the sketches. Note, however, that the output of the algorithm only depends on the “geometry” of the sketch, namely the bitwise length of the keys and on the specific hash functions used to populate it. In fact, a subset of the keys returned by the algorithm may not even be present in the sketch and represents false positive. A further confirmation stage is therefore needed to check *which keys* have been inserted in the sketch. This stage can be efficiently implemented by inserting the keys into a simple *footprint* Bloom filter at the same time they are inserted in the sketch.

V. THE PROBABILISTIC COUNTING FRAMEWORK

As discussed above, the counting framework proposed in this research is “doubly probabilistic”, being based on the combination of a probabilistic data structure and of a probabilistic counter. At the high level, the data structure is a three-dimensional array $S_{D \times W \times L}$, where each row d ($d = 1, \dots, D$) is associated with a function δ_d that takes values in the interval $(1, \dots, W)$ that are associated with the columns of the array.

Note that the two-dimensional substructure $S_{D \times W}$ is a *standard* reversible sketch table. The third dimension L is introduced to integrate the LogLog counting algorithm in such a sketch table. To this aim, each bucket of the sketch table is associated with another hash function H_{dw} that gives output in the interval $(1, \dots, L)$, associated with the layer (depth) of the array. In the construction phase, we have chosen to use hash functions belonging to the 4-universal hash family⁴ [57], obtained as:

$$h(x) = \sum_{i=0}^3 a_i \cdot x^i \mod p \mod W \quad (18)$$

where the coefficients a_i are arbitrarily chosen in the set $\{0, 1, \dots, p-1\}$ and p is a random prime number (we used the Mersenne numbers). Updating the data structure requires

⁴A class of hash functions $H : (1, \dots, N) \rightarrow (1, \dots, W)$ is a *k-universal hash* if for any distinct $x_0, \dots, x_{k-1} \in (1, \dots, N)$ and any possible $v_0, \dots, v_{k-1} \in (1, \dots, W)$:

$$Pr_{h \in H} \{h(x_i) = v_i; \forall i \in (1, \dots, k)\} = \frac{1}{W^k}$$

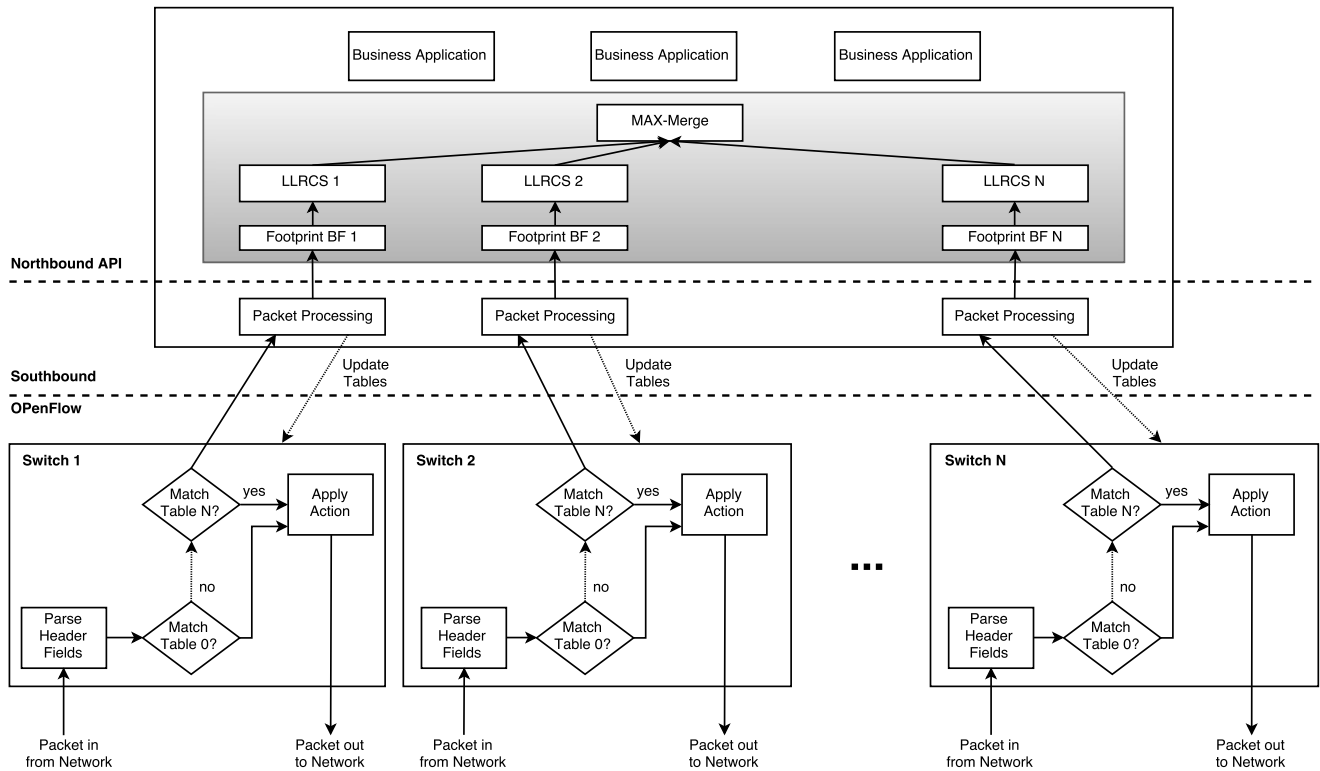


FIGURE 4. SDN-IXP counter of different flows.

first to choose the key of interest and apply mangling and hashing techniques to real data (to be counted) to select a bucket $S[d][w][\cdot]$ and the associated hash function H_{dw} . For all the keys that collide in a given bucket, the system computes the flow ID, typically given by a function of the header fields IP source and destination addresses, source and destination ports, and protocol. Then, it computes the hash function H_{dw} of the flow ID and updates the LogLog counter according to the LogLog counting algorithm.

Depending on the specific applications – this is the case of both the network applications addressed in this section – there might be the need of combining several data structures as a result of multiple instances of traffic measurements. More formally, each counter of the sketch represents the cardinality of the support of a multiset. When measurements are obtained at different points, and the results combined, each bucket ends up representing the sum of multisets (namely, a multiset that accommodates the items of each multiset together with the sum of their multiplicity) whose support is the union of the supports of all multisets. It is easy to convince ourselves that merging the data structures by means of the max-merge algorithm:

$$M[d][w][l] = \max_p S^p[d][w][l] \quad (19)$$

gives an estimation for the cardinality of the support set, that is the number of distinct elements of the aggregate multiset. Obviously, to allow this operation, all the different sketches

of counters must have been constructed by using the same hash functions.

Depending on the applications, a parallel data structure representing a reversible sketch (RS) may be constructed at each measurements point for identification purposes. Once again, RS must share common hash functions to apply the reverse algorithm.

It is important to highlight that such a framework must be *structured* in a different way, depending on the considered application scenario, especially considering distributed or centralised approaches. For this reason, in the following subsections, we discuss two use-cases (the same already introduced in Section III) where the framework will be respectively used to realise a centralised architecture for counting flows and a distributed architecture for detecting heavy hitters.

A. COUNTING UNKNOWN FLOWS IN SDN-BASED IXPS

The first application of the general purpose probabilistic counters described in the previous section refers to the SDN-IXP use-case. More precisely, the problem is to realize a monitoring application that estimates the number of different unknown flows in an SDN network. As shown in Figure 1, our application is developed as a business application on top of the SDN controller. The overall scheme of the network application is shown in Figure 4.

After receiving a packet for which there is no corresponding flow table entry, the switch sends such a packet to the controller, through the OpenFlow protocol.

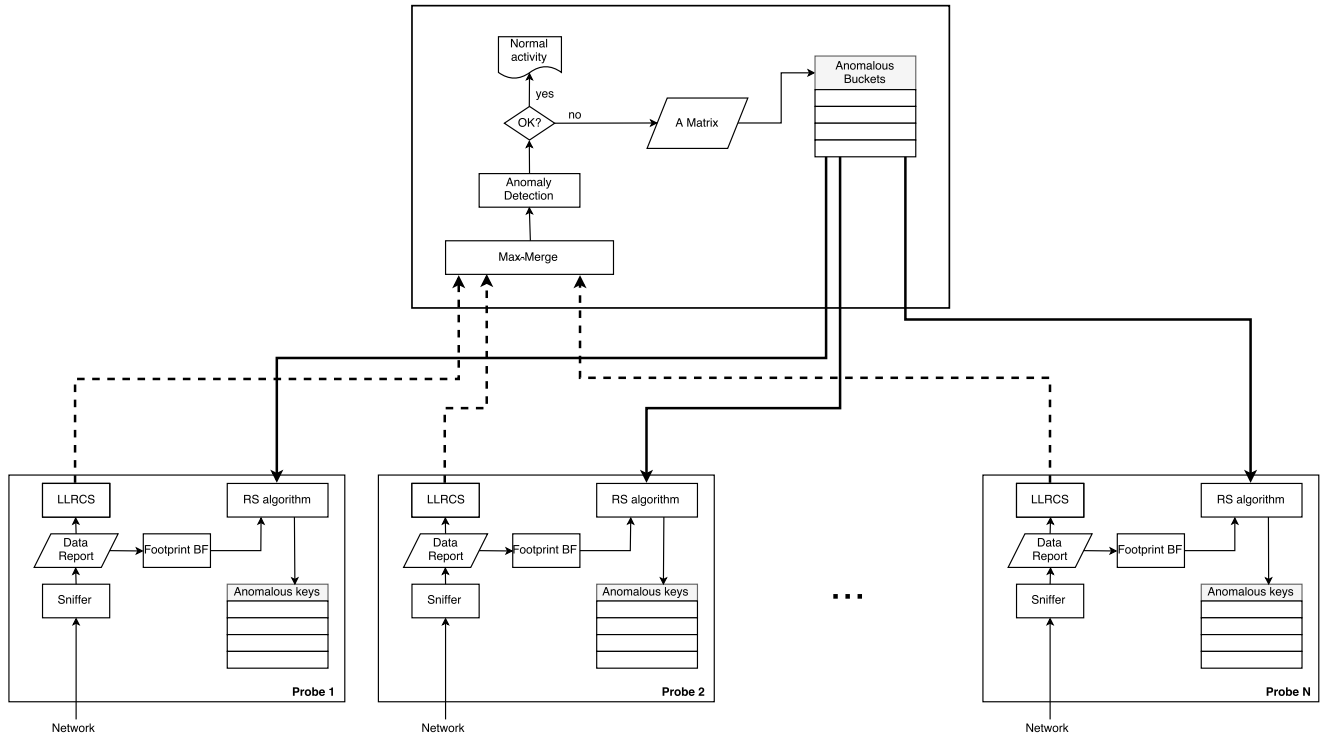


FIGURE 5. Anomaly detection application.

Hence, upon the reception of the packet, the controller first processes the packet and sets a new entry in the switch flow table by using OpenFlow, then passes the packet to the counter application that inserts the packet flow key in a footprint Bloom filter and finally updates the LogLog Counting Reversible Sketch table (LLCRS) corresponding to the switch that sent the packet (the reason why the controller must keep a distinct table for each switch will be clear in the following).

Once all the LLCRSs corresponding to the different switches have been constructed, they are merged through (19).

At this point, by applying equation (11) to each bucket $M[d][w][\cdot]$, the controller has an estimate of the “unknown” flows that collide in the same bucket. If needed (e.g., in case they exceed some given threshold), the controller can identify the flows by applying the reversible sketch algorithm and checking its output over the footprint Bloom filter.

Notice that such an apparently complex architecture to solve the problem of counting the “unknown” flows in an SDN network, is needed for several reasons, as discussed in the following points.

- The potentially high number of unknown flows in a network makes the use of deterministic data structure unfeasible; hence the sketch family data structure represents an optimal choice for such a kind of a problem.
- The use of a *standard* sketch for counting the flows (e.g., a count-min sketch [54]) is not satisfactory in our scenario, since there is the need of automatically discarding duplicates (i.e., the same flow traversing more switches).

- The use of a reversible sketch is justified by the fact that, once identified the unknown flows, the OpenFlow protocol allows to modify the behavior of the switch traversed by such flows.
- The choice of computing a distinct LLCRS for each distinct switch is needed to isolate the network segment in which the unknown flows are observed. Note that, in case this is not strictly necessary, the controller can simply compute a single LLCRS, skipping the max-merge phase.

It is important to highlight that in several domains (and quite likely in IXP/SDN scenarios), to solve the scalability issues typical of SDN networks, more than a single controller, organized hierarchically, are used. It is clear that our solution can easily be adapted to such a scenario, by observing that the LLCRS computed by a controller can be sent to a higher hierarchical level controller that aggregates the LLCRSs received by all the lower level controllers.

B. PROBABILISTIC COUNTING FOR ANOMALY DETECTION IN BACKBONE NETWORKS

This section presents the application of the probabilistic counting framework to the anomaly detection use-case by showing the scheme of a distributed and collaborative Intrusion Detection system. The overall architecture of the application is shown in Figure 5.

Multiple detection probes distributed in the network monitor separate portions of the network itself and report the collected information to a single location (mediator) in charge of processing data and raise alerts in case of suspicious traffic

activity. A backtrack mechanism is then used by the mediator to ask the probes for flow identification.

Starting from the observed traffic, each probe produces a periodic report that contains information related to the traffic measured in a given time bin, which consists of a collection of flow keys: $\langle IP \text{ source address, } IP \text{ destination address, source Port, destination Port, Protocol} \rangle$.

The periodic reports are passed as input to the module responsible for the construction and update of the LogLog Counting Reversible Sketch as well as the footprint Bloom filter. At this stage, each probe has a summary of all the different observed flows, together with an estimate of their number.

The mediator, responsible for the detection phase, combines the information exported by each probe through the equation (19).

Since the max-merge operation implicitly solves the problem of not counting duplicated flows, the resulting aggregated sketch is precisely equivalent to the one that would be constructed in an “ideal” case, in which the mediator would be able to observe all the traffic, directly. This property also implies that the estimation error, due to the probabilistic nature of the data structure, is not worsened by the distributed nature of the application, being equivalent to that of a single probabilistic counter.

At this point, the mediator counts the number of distinct flows that collide in the same bucket. This task can be solved by applying equation (11) to each bucket $M[d][w][\cdot]$.

At this stage, by using a classical detection method (e.g., PCA, wavelet analysis, heavy hitter) the system can decide if there are or not anomalous aggregates in a given time bin. The output of this phase is a binary matrix $A[d][w]$ that contains a “1” if the corresponding bucket is considered anomalous “0”, otherwise.

Note that, given the nature of the sketches, each traffic flow is part of several random aggregates (namely D aggregates), corresponding to the D different hash functions. This means that, in practice, any flow will be checked D times to verify if it presents an anomaly (this is done because an anomalous flow could be masked in a given traffic aggregate, while being detectable in another one).

Due to this fact, a voting algorithm is applied to the matrix A . The algorithm verifies if at least H rows of A contain at least a bucket set to “1” (H is a tunable parameter). If so the mediator reveals an anomaly, otherwise, the matrix A is discarded.

In case the mediator reveals some anomalous time bin during the detection phase, it back-propagates the matrix $A[d][w]$ to the probes.

At this point, each probe uses the RS computed for the anomalous time bin and the footprint Bloom filter for identifying the IP addresses responsible for the anomalies.

C. SPACE/TIME COMPLEXITY ANALYSIS

The proposed counting framework involves the execution of several operations on the data structures.

As far as *space requirements*, the memory footprint of the system depends on the target error rate σ and it is given by $O(\sigma^{-3})$, roughly due to the composition of the LogLog counter complexity $O(\sigma^{-2})$ into the sketch complexity $O(\sigma^{-1})$.

Regarding *time requirements*, the overall complexity is given by the combination of elementary operations, which may occur depending on the specific use-case. The cost of *update* in each sketch is given by $O(1)$ (bucket selection) and $O(1)$ for the add operation of the counter (we consider the HyperLogLog in this section). Then, merging the sketches involves the computation of a bucket-wise maximum over N sketches. As long as N , number of monitoring probes, is a constant, the cost of such operation is $O(1)$. Finally, the *query* complexity is $O(1)$ (bucket selection) and $O(1)$ is the complexity of the *count* operation of the counter as well.

VI. C++ IMPLEMENTATION

The probabilistic framework has been implemented in modern C++ within the *pds* (*probabilistic data structures*) library and is freely available for download at [58]. The library includes the implementation of some randomized data structures, such as Bloom filters, Counting Bloom filters, Sketches, LogLog counters (and their variants), as well as a set of utility functions, algorithms, and tests.

The library is designed according to the following principles:

- *Generic Programming and Composability*. Overall, the library pervasively uses templates to ease code reusability. All data structures are indeed abstract and implemented as containers of generic data that can be specialized upon need. As an example, the sketch buckets may contain any data (e.g., integer, arrays, LogLog counters, etc.) equipped with proper operations.
- *Robustness*. The inherent complexity of managing advanced data structures may easily lead to configuration errors (e.g., types of data, hash functions codomain bitwise length, etc.). The use of template meta-programming techniques allows to spot such errors at compile-time and enforce the correctness and the consistency of the declared parameters using static asserts.
- *Declarative syntax*. The properties of all data structures are embedded within their declaration, as it improves the usability of the library. As an example, a sketch declaration consists of the size of the sketch itself, the type of data accommodated in the buckets, the hash functions used in each row together with their codomain bitwise dimensions.

The following subsections present more details about the implementation of reversible sketches and LogLog counters as they are used in this work. The signatures of the most significant methods are also explicitly reported as a reference to help interested readers in using the library.

A. LogLog COUNTERS

The use of a LogLog counter `ll` is simply instantiated as follows:

```
using LogLog_type =
    pds::LogLog< uint8_t
                , 1024
                , std::hash<std::string>>;
LogLog_type ll;
```

The statement declares a LogLog counter suitable for estimating the cardinality of a multiset of strings. The number of “small bytes” is set to 1024 while their type is an unsigned byte (although 5 bits are sufficient). In the example, the strings are hashed through the standard hash function for strings (FNV), though any other function can be used as the parameter of the template definition.

Hyper LogLog counters are also implemented in the library and its use only requires replacing the name `hyperLogLog_type` to `LogLog_type` in the above declaration.

B. SKETCHES

A simple example of sketch declaration for string counting is the following:

```
using sketch_type =
    pds::sketch< uint32_t
                , 1024
                , BIT_10(myhash1<string>)
                , BIT_10(myhash2<string>>>;
sketch_type s;
```

The `using` statement declares a sketch of 1024 columns and two rows, each with a different user-defined hash function. The `BIT_10` macros are herein used to annotate that the co-domain bit-size of such hash functions is set to 10. At compile-time, suitable meta-functions evaluate the hash sizes and verify the consistency with the sketch size.

The class provides a broad number of iterators that allows visiting the buckets (one per line) in correspondence of a specific element to be inserted in the sketch. For instance, the method:

```
s.foreach_bucket("fortytwo", [](int &bkt) {
    bkt++;
});
```

iterates over the two buckets associated with the string `fortytwo` and updates the bucket content using the given lambda function. For standard operations like increment or decrement, methods like `increment_buckets` and `decrement_buckets` are also provided.

Besides, the sketch class provides the count-min estimation, the k-ary estimation, and a set of more general utility functions for the sketch management, including filtering buckets, searching elements and sketches aggregation.

C. REVERSIBLE SKETCHES

The reverse sketch algorithm [56] requires the bitwise partitioning of the keys (elements) used to update the sketch and that each part of the keys (word) is hashed separately. In the library abstraction, elements are then represented as *tuples* of generic types in which each component of the tuple represents a portion of the original key. The modular hash is then applied to the tuple as a whole by concatenating the single hash functions applied to each component of the tuple. This representation is particularly convenient in networking use-cases as standard keys come from the concatenation of different packet header fields, such as IP addresses, TCP ports, protocol field, and so on. Notice that, even single field keys (e.g., IP addresses), can be conveniently split into several parts to improve the efficiency of the algorithm.

As an example, the C++ key for a TCP flow can be represented as:

```
auto key = std::make_tuple( ip_src.high
                             , ip_src.low
                             , ip_dst.high
                             , ip_dst.low
                             , src_port
                             , dst_port
                             , proto );
```

where source and destination IP addresses are broken into two 16 bits long portions.

The library provides the implementation of a modular hash for generic N component tuples, obtained by composing N different hash subfunctions. Each subfunction is applied to a single component of the tuple; the final result is obtained by concatenating the output of all subfunctions. Notice that both the bitwise length of the tuple components as well as the bitwise length of the output of the hash subfunction is fully configurable at compile-time.

The following declaration defines a modular hash function for the above-presented TCP flow tuple.

```
using hash_type = pds::ModularHash< BIT_4(H1)
                                     , BIT_4(H1)
                                     , BIT_4(H1)
                                     , BIT_4(H1)
                                     , BIT_3(H2)
                                     , BIT_3(H2)
                                     , BIT_3(H3) >
```

Three different functions $H1$, $H2$, and $H3$ are used (but a single function could have been used as well). The output of $H1$ is constrained to be 4 bits long, while $H2$ and $H3$ produce 3 bits long results. Overall, the complete hash functions yields a 25 bits long output and is obviously represented as a 64 bit integer.

Therefore, a reversible sketch `s` that uses the canonical IP flow can be defined as follows (for the sake of simplicity, the same hash function is applied to all the components of the tuple):

```

pds::sketch< uint16_t
    , (1<< 21)
    , pds::ModularHash< BIT_4(H1)
                        , BIT_4(H1)
                        , BIT_4(H1)
                        , BIT_4(H1)
                        , BIT_3(H1)
                        , BIT_3(H1)
                        , BIT_3(H1) >
    , pds::ModularHash< BIT_4(H2)
                        , BIT_4(H2)
                        , BIT_4(H2)
                        , BIT_4(H2)
                        , BIT_3(H2)
                        , BIT_3(H2)
                        , BIT_3(H2) >
    , pds::ModularHash< BIT_4(H3)
                        , BIT_4(H3)
                        , BIT_4(H3)
                        , BIT_4(H3)
                        , BIT_3(H3)
                        , BIT_3(H3)
                        , BIT_3(H3) >
    , pds::ModularHash< BIT_4(H4)
                        , BIT_4(H4)
                        , BIT_4(H4)
                        , BIT_4(H4)
                        , BIT_3(H4)
                        , BIT_3(H4)
                        , BIT_3(H4) >
    , pds::ModularHash< BIT_4(H5)
                        , BIT_4(H5)
                        , BIT_4(H5)
                        , BIT_4(H5)
                        , BIT_3(H5)
                        , BIT_3(H5)
                        , BIT_3(H5) >
    , pds::ModularHash< BIT_4(H6)
                        , BIT_4(H6)
                        , BIT_4(H6)
                        , BIT_4(H6)
                        , BIT_3(H6)
                        , BIT_3(H6)
                        , BIT_3(H6) >
> s;

```

Sketch updates are made through the `increment_buckets` methods. The following code emulates the TCP flow:

```
< 0xbad, 0xbec, 0xdead, 0xbeef, 6010, 4216, 6 >
```

and the UDP flow:

```
< 0xdead, 0xbeef, 0xcafe, 0xbabe, 80, 6667, 17 >
```

both hitting the sketch 1000 times and triggering the increment of the associated buckets, accordingly.

```

for(auto n = 0; n < 1000; n++)
{
    s.increment_buckets(std::make_tuple(
        0xbad, 0xbec, 0xdead, 0xbeef, 6010, 4216, 6)

    s.foreach_bucket("fortytwo", [](int &bkt) {
        bkt++;
    });
}

```

The method `index_buckets` is used to compute a predicate (represented as a *lambda function*) over the whole sketch. In the following example, the method is used to retrieve the buckets whose content exceeds 500. The result

is a vector of vectors that contains the indexes of the bucket (per each row of the sketch) that satisfy the predicate.

```

auto idx = s.index_buckets([](auto &b)
{
    return b > 500;
});

```

The final step is to feed the method `reverse_sketch` with the above-obtained bucket indexes to obtain the list of tuples (i.e., keys) that hit the sketch in the selected buckets. The following code is used to reverse the sketch.

```

auto rev = pds::reverse_sketch< uint16_t
                                , uint16_t
                                , uint16_t
                                , uint16_t
                                , uint16_t
                                , uint16_t
                                , uint8_t>(s, idx);

```

D. LogLog COUNTING REVERSIBLE SKETCHES

The LogLog counting reversible sketch is obtained by wrapping up the previously described components and defining each bucket of the sketch to be a LogLog counter.

The following example refers to the simple case of a *port scan detector*. In this scenario, the attacker is an Internet host that sends packets with different destination port numbers to another host to check for open ports.

The sketch is then populated by hashing on the pair source and destination IP addresses that, in the example, are both conveniently split into two parts. The LogLog counter, instead, is updated by using the pair source/destination ports and will only get incremented upon ports variation.

```

using LogLog_t =
    pds::hyperLogLog< uint8_t
                    , 64
                    ,
std::hash<std::tuple<uint16_t, uint16_t>>
                    >;
pds::sketch< LogLog_t
    , (1 << 16)
    , pds::ModularHash< BIT_4(H1)
                        , BIT_4(H1)
                        , BIT_4(H1)
                        , BIT_4(H1) >
    , pds::ModularHash< BIT_4(H2)
                        , BIT_4(H2)
                        , BIT_4(H2)
                        , BIT_4(H2) >
    , pds::ModularHash< BIT_4(H3)
                        , BIT_4(H3)
                        , BIT_4(H3)
                        , BIT_4(H3) >
    , pds::ModularHash< BIT_4(H4)
                        , BIT_4(H4)
                        , BIT_4(H4)
                        , BIT_4(H4) >
    , pds::ModularHash< BIT_4(H5)
                        , BIT_4(H5)
                        , BIT_4(H5)
                        , BIT_4(H5) >
> s;

```

The method `cardinality` is used to retrieve the index of the LogLog counters with values bigger than 10000.

```
auto idx = s.index_buckets([](auto &b)
{
    return b.cardinality() > 10000;
});
```

Likewise, the list of candidates is obtained by reverting the sketch through the following code:

```
auto rev = pds::reverse_sketch< uint16_t
                                , uint16_t
                                , uint16_t
                                , uint16_t
                                > (s, idx);
```

VII. EXPERIMENTAL RESULTS

The doubly probabilistic nature of the overall counting framework introduces two possible levels of errors whose effects are not always easily predictable in practice. The first obvious source of uncertainty is given by the statistic nature of the counter for which, however, theoretical bounds are available. The second – and more subtle – source of errors is instead given by the collisions that may occur when populating the sketch. Indeed, if erroneously detected keys can be readily discarded by a simple footprint Bloom filter, the effect of collisions in the counter updating mechanisms cannot be depurated and requires the sketch size to be thoroughly tuned according to the application requirements to avoid measurement corruption.

In this section we present the performance assessment of the proposed system in the pretty general case of the *heavy hitter* detector application. The application itself can run on a single vantage point or in a distributed fashion and is a particular case of the anomaly detection algorithm presented in section V-B. Upon receiving a packet, the probes involved in the measurement extract the source IP address and use it as the key to increment the corresponding buckets of its LLRCS sketch. Each probe periodically sends the LLRCS up to the mediator, which merges the received LLRCSs and checks if the traffic volume recorded in some buckets exceeds a given threshold (typically expressed regarding a percentage of the total recorded traffic load). In the affirmative case, the mediator sends the address of the “anomalous” bucket back to the probes which, in turn, run the reverse algorithm and refine the results by the footprint Bloom filter to come up with the responsible *heavy hitters* (IP addresses).

Although quite simple, such a use-case involves a complete set of critical parameters that need to be investigated to assess the performance of the whole system. More in details, the following performance indexes will be investigated:

- Sketch size
- False alarm rate

All of the parameters are strictly correlated and point directly to the primary motivation of this work. Indeed, we advocated the use of our system because of the

TABLE 1. Compression factor.

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	603.0976	301.5488	150.7744	75.3872
10	150.7744	75.3872	37.6936	18.8468
12	37.6936	18.8468	9.4234	4.7117
14	9.4234	4.7117	2.35585	1.177925
16	2.35585	1.177925	0.5889625	0.29448125

compactness of the data structures that, in turn, facilitate their transfer and their storage in small memory devices. However, the size of the data structure is determined by two factors: i) the size of the sketch (namely, the number of rows, 4 in our tests, and columns, determined by the length of the hash functions) and ii) the depth of the counter (i.e., the size of the small bytes). Obviously, the smaller the data structure, the higher the false positive rate (due to the more substantial number of collisions) and the lower the estimation accuracy (due to the smaller size of the counters).

In all experiments, the application ran in stand-alone mode as the distributed behavior would not add any valuable insights to the performance analysis. Hence, a single probe was fed with a real trace from the MAWI repository [59], containing 928223 distinct flows and 260851 distinct IP source addresses, corresponding to 15 minutes of traffic collected at 2PM of September 28, 2016. Also, a trace containing a synthetic heavy hitter was mixed in case the original trace would not include any high hitting flow to detect.

Starting with the analysis of the sketch size, in Table 1 we present the compression factor achieved by the sketch in comparison with a C++ unordered map (containing the exact counters). We can easily see that the compression rate obviously decrease when increasing the hash output length (number of columns) or the LogLog bucket size. Such results are not significant by themselves, since as already stated in the previous sections the bigger the sketch, the better the performance of the counters, and must be commented together with the following ones.

Hence, moving to the False Alarm rate, it is first of all worth highlighting that it depends on three distinct components:

- post-filtering phase performed by means of the *footprint* Bloom filter (as explained in Section IV-B)
- number of collisions
- count estimation error

Regarding the first one, Tables 2 and 3 respectively show the performance achieved with and without the use of the *footprint* Bloom filter. In our experiments the BF length m has been set according to the optimal size of $m = n * k / \log(2)$, where k is the number of hash functions (4 in our case), and n is the expected number of flows, rounded up to the smallest power of 2 (512 KByte in our tests).

It is easy to understand that the post-filtering phase only affects the performance in case of “short” hash functions and is negligible in the other cases. Indeed, the performance

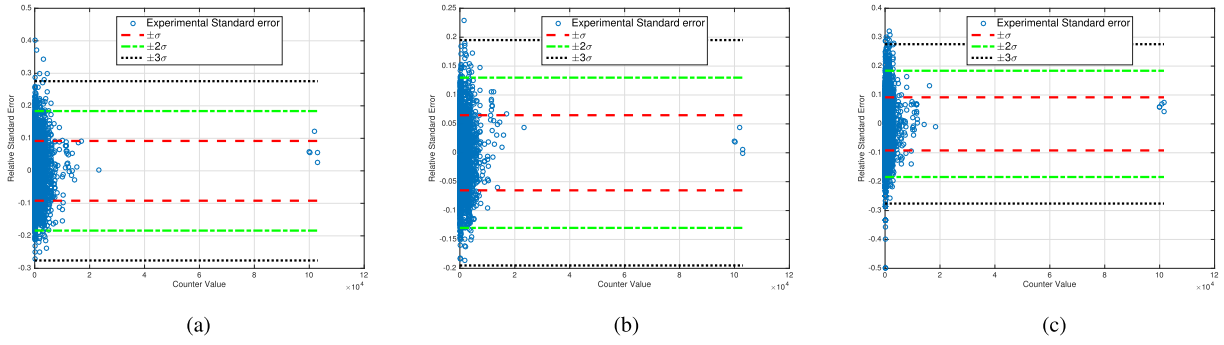


FIGURE 6. Count error estimation in three reasonable cases. (a) Hash length = 10 – Bucket size = 128. (b) Hash length = 10 – Bucket size = 256. (c) Hash length = 12 – Bucket size = 128.

TABLE 2. False alarm rate (%).

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	65.38	66.38	66.3777	67.09
10	0.03	0.023	0.017	0.013
12	0.0023	0.0027	0.0034	0.0031
14	0.0019	0.00157	0.0031	0.0027
16	0.0016	0.00157	0.0019	0.0027

TABLE 3. False alarm rate (%) - without footprint BF.

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	99.99994	99.99994	99.99994	99.99994
10	0.16369	0.12267	0.07629	0.06670
12	0.00230	0.00268	0.00345	0.00306
14	0.00191	0.00153	0.00306	0.00268
16	0.00153	0.00153	0.00191	0.00268

TABLE 4. Count estimation within $\pm\sigma$.

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	0.721875	0.726562	0.735156	0.746094
10	0.726758	0.749805	0.777734	0.805469
12	0.811667	0.827997	0.847474	0.857016
14	0.889683	0.906507	0.924965	0.934547
16	0.92347	0.937113	0.949509	0.949509

is significantly improved for all of the cases corresponding to 8 and 10. It is worth noticing that in the cases 8, the use of the BFs is of primary importance. Indeed, without such filter, not only would the system reveal almost all of the observed flows as potential candidates, but it would also indicate a large number of unobserved flows as potential candidates (this is due to the nature of the RS algorithm).

Moving to the impact of the collisions and the estimation error on the false alarm rate, it is essential to specify that it is not possible to precisely analyze their impact separately. Nonetheless, it is very intuitive that collisions depend on the hash length, while the estimation error depends on the bucket size, as clearly demonstrated by the results shown in Table 2. To better evaluate the Count estimation error,

TABLE 5. Count estimation within $\pm 2\sigma$.

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	0.957812	0.9625	0.964063	0.975
10	0.964063	0.97207	0.981055	0.988477
12	0.981555	0.984064	0.981063	0.977915
14	0.986909	0.986333	0.983464	0.983577
16	0.989282	0.988879	0.987806	0.988549

TABLE 6. Count estimation within $\pm 3\sigma$.

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	0.989062	0.99375	0.996875	0.999219
10	0.993359	0.996289	0.998633	0.999805
12	0.998033	0.998574	0.998229	0.995672
14	0.999327	0.99686	0.995834	0.993767
16	0.999377	0.996993	0.996048	0.995141

TABLE 7. Space/time requirements in real settings.

Hash Length	Bucket Size	Time (msec)	Space (MB)
10	128	68.54	0.625
10	256	63.95	1.25
12	128	5.29	2.5

in Tables 4, 5, and 6 we show the percentage of flows that are in within $\pm\sigma$, $\pm 2\sigma$ and $\pm 3\sigma$ of the exact count. As expected we can see that in all of the cases the constraints (described in Section IV-A) are met. Such results are also visually depicted in Figure 6, where we show three reasonable cases.

To give an idea of the actual time/space complexity in real world settings, Table 7 shows the memory footprint and the execution time of the algorithm running on a general purpose PC equipped with an Intel i7-2600 CPU at 3.4Ghz and 8GB of RAM for the three above cases.

Summing up, by inspecting the different performance figures, it is possible to select a set of cases that offer the best trade-off between memory footprint and detection probability. In general, all of the cases corresponding to hash length of 10 and 12 bits provide acceptable performance and the choice among them can be driven by memory

availability. Indeed, taking as an example the case with the hash length of 10 bits and bucket size of 128 bits, the system presents a false alarm rate of 0.017% with a compression factor of around 37.7, which make the data structure rather thin while still providing excellent detection performance.

VIII. CONCLUSION

The paper presents a generic counting framework for the estimation of the cardinality of large multi-sets. The framework is based on the combined use of reversible sketches and LogLog counters in a fast and compact data structure that can be suitably integrated for streaming processing in the data-plane of network devices. Two different use-cases from the monitoring and the security domains are also described. A full software implementation of the proposed framework is reported within the probabilistic data structures (*pds*) library. The library is built upon a generic programming model and template technology so as to be easily specialized upon need, even in totally different contexts. The system as a whole, along with its software implementation, has been experimentally validated and the *pds* library released for download as open-source.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for the precious comments and suggestions.

REFERENCES

- [1] Cisco Systems. (Jun. 2017). *Cisco Visual Networking Index: Forecast and Methodology*. [Online]. Available: <http://www.cisco.com>
- [2] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. ACM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, New York, NY, USA, 2002, pp. 270–313. doi: 10.1145/633025.633056.
- [3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.
- [4] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi, "Implementing advanced network functions for datacenters with stateful programmable data planes," in *Proc. IEEE Int. Symp. Local Metropolitan Area Netw. (LANMAN)*, Jun. 2017, pp. 1–6.
- [5] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
- [6] N. Bonelli, S. Giordano, and G. Prociassi, "Network traffic processing with PFQ," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 6, pp. 1819–1833, Jun. 2016.
- [7] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices* (The Morgan Kaufmann Series in Networking). San Francisco, CA, USA: Morgan Kaufmann, 2004.
- [8] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. ESA*, 2003, pp. 605–617.
- [9] G. Bianchi et al., "Privacy-preserving network monitoring: Challenges and solutions," in *Proc. 17th ICT Mobile Wireless Commun. Summit*, 2008, pp. 1–10.
- [10] C. Callegari, A. Di Pietro, S. Giordano, T. Pepe, and G. Prociassi, "The loglog counting reversible sketch: A distributed architecture for detecting anomalies in backbone networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2012, pp. 1287–1291.
- [11] C. Callegari, S. Giordano, M. Pagano, and G. Prociassi, "OpenCounter: Counting unknown flows in software defined networks," in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst., SummerSim Multiconf. (SPECTS)*, 2015, pp. 1–7.
- [12] A. Dainotti, A. Pescapè, and K. C. Claffy, "Issues and future directions in traffic classification," *IEEE Netw.*, vol. 26, no. 1, pp. 35–40, Jan./Feb. 2012.
- [13] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. ACM 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, New York, NY, USA, 2014, pp. 61–66. doi: 10.1145/2620728.2620729.
- [14] N. Bonelli, S. Giordano, and G. Prociassi, "A pipeline functional language for stateful packet processing," in *Proc. IEEE Conf. Netw. Softw., Softw. Sustaining Hyper-Connected World, Route 5G (NetSoft)*, Jul. 2017, pp. 1–4.
- [15] N. Bonelli, S. Giordano, and G. Prociassi, "Enif-lang: A specialized language for programming network functions on commodity hardware," *J. Sens. Actuator Netw.*, vol. 7, no. 3, p. 34, 2018.
- [16] N. Bonelli, S. Giordano, G. Prociassi, and L. Abeni, "A purely functional approach to packet processing," in *Proc. 10th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oct. 2014, pp. 219–230.
- [17] *In-Band Network Telemetry*. Accessed: Feb. 2019. [Online]. Available: <https://p4.org/p4/inband-network-telemetry/>
- [18] *In Situ OAM*. Accessed: Feb. 2019. [Online]. Available: <https://datatracker.ietf.org/wg/ioam/about/>
- [19] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implementation. (NSDI)*, Berkeley, CA, USA: USENIX Association, 2013, pp. 29–42. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2482626.2482631>
- [20] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, 1999.
- [21] J.-B. Tristan, J. Tassarotti, and G. L. Steele, Jr., "Efficient training of LDA on a GPU by mean-for-mode estimation," in *Proc. 32nd Int. Conf. Mach. Learn.*, vol. 37, Lille, France, F. Bach and D. Blei, Eds., Jul. 2015, pp. 59–68. [Online]. Available: <http://proceedings.mlr.press/v37/tristan15.html>
- [22] G. L. Steele, Jr., and J.-B. Tristan, "Adding approximate counters," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2016, pp. 15–1–15–12. doi: 10.1145/2851141.2851147.
- [23] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high-speed links," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 925–937, Oct. 2006. doi: 10.1109/TNET.2006.882836.
- [24] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi, "Fast data stream algorithms using associative memories," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2007, pp. 247–256. doi: 10.1145/1247480.1247510.
- [25] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *Proc. Very Large Data Bases Conf. (VLDB)*, 2003, pp. 464–475.
- [26] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas. (IMC)*, New York, NY, USA, 2004, pp. 101–114. doi: 10.1145/1028788.1028802.
- [27] A. Kumar, M. Sung, J. J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *Proc. ACM Perform. Joint Int. Conf. Meas. Modelling Comput. Syst. (SIGMETRICS)*, New York, NY, USA, 2004, pp. 177–188. doi: 10.1145/1005686.1005709.
- [28] J. Sanjuàns-Cuxart, P. Barlet-Ros, N. Duffield, and R. R. Kompella, "Sketching the delay: Tracking temporally uncorrelated flow-level latencies," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf. (IMC)*, New York, NY, USA, 2011, pp. 483–498. doi: 10.1145/2068816.2068861.
- [29] G. Huang, A. Lall, C.-N. Chuah, and J. Xu, "Uncovering global icebergs in distributed streams: Results and implications," *J. Netw. Syst. Manage.*, vol. 19, no. 1, pp. 84–110, Mar. 2011. doi: 10.1007/s10922-010-9186-5.
- [30] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 323–334, Jun. 2006. doi: 10.1145/1140103.1140314.
- [31] R. Stanojevic, "Small active counters," in *Proc. 26th IEEE Int. Conf. Comput. Commun. (IEEE INFOCOM)*, May 2007, pp. 2153–2161.
- [32] C. Hu et al., "DISCO: Memory efficient and accurate flow statistics for network measurement," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 665–674.
- [33] E. Tsidon, I. Hanniel, and I. Keslassy, "Estimators also need shared values to grow together," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 1889–1897.

- [34] G. Einziger, B. Fellman, and Y. Kassner, "Independent counter estimation buckets," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 2560–2568.
- [35] Y. Li, H. Wu, T. Pan, H. Dai, J. Lu, and B. Liu, "CASE: Cache-assisted stretchable estimator for high speed per-flow measurement," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2016, pp. 1–9.
- [36] Y. Afek, A. Bremner-Barr, S. L. Feibish, and L. Schiff, "Detecting heavy flows in the SDN match and action model," *Comput. Netw.*, vol. 136, pp. 1–12, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1389128618300859>
- [37] R. B. Basat, G. Einziger, S. L. Feibish, J. Moroney, and D. Raz, "Network-wide routing-oblivious heavy hitters," in *Proc. ACM Symp. Archit. Netw. Commun. Syst. (ANCS)*, New York, NY, USA, 2018, pp. 66–73. doi: [10.1145/3230718.3230729](https://doi.org/10.1145/3230718.3230729).
- [38] L. Deri. *PF_RING ZC (Zero Copy)*. Accessed: Feb. 2019. [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/
- [39] L. Rizzo, "NetMap: A novel framework for fast packet I/O," in *Proc. USENIX ATC*. Berkeley, CA, USA: USENIX Association, 2012, pp. 101–112.
- [40] *DPDK*. Accessed: Feb. 2019. [Online]. Available: <http://dpdk.org>
- [41] A. Gupta et al., "SDX: A software defined Internet exchange," in *Proc. ACM SIGCOMM*, 2014, pp. 551–562.
- [42] J. P. Stringer, Q. Fu, C. Lorier, R. Nelson, and C. E. Rothenberg, "Cardigan: Deploying a distributed routing fabric," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, New York, NY, USA, 2013, pp. 169–170. doi: [10.1145/2491185.2491221](https://doi.org/10.1145/2491185.2491221).
- [43] M. Jager, "Securing IXP connectivity," in *Proc. APNIC*, 2012, pp. 1–41.
- [44] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proc. Internet Meas. Workshop*, 2002, pp. 71–82.
- [45] J. D. Brutlag, "Aberrant behavior detection in time series for network monitoring," in *Proc. 14th USENIX Conf. Syst. Admin.*. Berkeley, CA, USA: USENIX Association, 2000, pp. 139–146. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1045502.1045530>
- [46] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *ACM SIGCOMM*, 2004, pp. 219–230.
- [47] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan, "Network anomography," in *Proc. IMC*, 2005, p. 30.
- [48] M. Thottan and C. Ji, "Anomaly detection in IP network," *IEEE Trans. Signal Process.*, vol. 51, no. 8, pp. 2191–2204, Aug. 2003.
- [49] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Comput. Secur.*, vol. 29, no. 1, pp. 124–140, 2010.
- [50] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, Oct. 1985. [Online]. Available: <http://portal.acm.org/citation.cfm?id=5212.5215>
- [51] W. Szpankowski and V. Rego, "Yet another application of a binomial recurrence order statistics," *Computing*, vol. 43, no. 4, pp. 401–410, 1990. doi: [10.1007/BF02241658](https://doi.org/10.1007/BF02241658).
- [52] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Anal. Algorithms (AOFA)*, 2007, pp. 128–146.
- [53] G. Cormode, T. Johnson, F. Korn, O. Spatscheck, D. Srivastava, and S. Muthukrishnan, "Holistic UDAFs at streaming speeds," in *Proc. SIGMOD*, 2004, pp. 35–46. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.2257>
- [54] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
- [55] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. 3rd ACM SIGCOMM Conf. Internet Meas. (IMC)*. New York, NY, USA: ACM Press, 2003, pp. 234–247. doi: [10.1145/948205.948236](https://doi.org/10.1145/948205.948236).
- [56] R. Schwellen, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. ACM SIGCOMM Conf. Internet Meas. (IMC)*, New York, NY, USA, 2004, pp. 207–212. doi: [10.1145/1028788.1028814](https://doi.org/10.1145/1028788.1028814).
- [57] M. Thorup and Y. Zhang, "Tabulation based 4-universal hashing with applications to second moment estimation," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*. Philadelphia, PA, USA: SIAM, 2004, pp. 615–624.
- [58] *Probabilistic Data Structure (PDS) Library*. Accessed: Feb. 2019. [Online]. Available: <https://github.com/awgn/pds>
- [59] MAWI Working Group. *Packet Traces From WIDE Backbone*. Accessed: Feb. 2019. [Online]. Available: <http://mawi.wide.ad.jp/mawi/>



NICOLA BONELLI received the master's degree in telecommunication engineering and the Ph.D. degree in information engineering from the Università di Pisa.

He collaborates with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni, where he was involved in the European Research project Behavioral-Based forwarding. His current research interests include functional languages, software defined networking, wait-free and lock-free algorithms, transactional data-structures, parallel computing, and concurrent programming (multi-threaded) on multi-core architectures.



CHRISTIAN CALLEGARI received the Laurea degree (*cum laude*) in telecommunication engineering from the Università di Pisa, in 2004, discussing a thesis titled Simulative analysis of RSVP-TE, and evaluation of end-to-end rerouting techniques in MPLS networks and the Ph.D. degree in information engineering from the Department of Information Engineering, Università di Pisa, in 2008. In 2005, he obtained the qualification to practice the profession of Engineer. He is currently a Researcher with the Radar and Surveillance Systems National Laboratory of CNIT. His research interests include network security, traffic classification, traffic engineering, MPLS architecture, and network simulation. He is a member of the IEEE Communication Society.



GREGORIO PROCIASSI received the graduate degree in telecommunication engineering and the Ph.D. degree in information engineering from the Università di Pisa, Pisa, Italy, in 1997 and 2002, respectively.

From 2000 to 2001, he was a Visiting Scholar with the Computer Science Department, University of California at Los Angeles, Los Angeles. In 2002, he became a Researcher with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni, Research Unit of Pisa. Since 2005, he has been an Assistant Professor with the Department of Information Engineering, Università di Pisa. He has worked in several research projects funded by NSF, DARPA, European Union, and Italian MIUR. His research interests include measurements, modeling, and performance evaluation of IP networks.

...