Generalized Sketch Families for Network Traffic Measurement

YOU ZHOU*, University of Florida, USA
YOULIN ZHANG*, University of Florida, USA
CHAOYI MA, University of Florida, USA
SHIGANG CHEN, University of Florida, USA
OLUFEMI O ODEGBILE, University of Florida, USA

Traffic measurement provides critical information for network management, resource allocation, traffic engineering, and attack detection. Most prior art has been geared towards specific application needs with specific performance objectives. To support diverse requirements with efficient and future-proof implementation, this paper takes a new approach to establish common frameworks, each for a family of traffic measurement solutions that share the same implementation structure, providing a high level of generality, for both size and spread measurements and for all flows. The designs support many options of performance-overhead tradeoff with as few as one memory update per packet and as little space as several bits per flow on average. Such a family-based approach will unify implementation by removing redundancy from different measurement tasks and support reconfigurability in a plug-n-play manner. We demonstrate the connection and difference in the design of these traffic measurement families and perform experimental comparisons on hardware/software platforms to find their tradeoff, which provide practical guidance for which solutions to use under given performance goals.

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**;

Additional Key Words and Phrases: Network Traffic Measurement;Big Network Data; Generalized Sketch Families

ACM Reference Format:
You Zhou, Youlin Zhang, Chaoyi Ma, Shigang Chen, and Olufemi O Odegbile. 2019. Generalized Sketch Families for Network Traffic Measurement. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article 51 (December 2019), 34 pages. https://doi.org/10.1145/3366699

1 INTRODUCTION

Traffic measurement is a critical function for network management, resource alignment, traffic engineering, and anomaly detection. In particular, flow size and flow spread provide fine-grained and general-purpose information from which a variety of traffic statistics can

*You Zhou and Youlin Zhang are co-first authors

Authors' addresses: You Zhou, youzhou@cise.ufl.edu, University of Florida, Gainsville, FL, USA; Youlin Zhang, ylzh10@ufl.edu, University of Florida, Gainsville, FL, USA; Chaoyi Ma, ch.ma@ufl.edu, University of Florida, Gainsville, FL, USA; Shigang Chen, sgchen@cise.ufl.edu, University of Florida, Gainsville, FL, USA; Olufemi O Odegbile, oodegbile@ufl.edu, University of Florida, Gainsville, FL, USA.

2476-1249/2019/12-ART51 $15.00
https://doi.org/10.1145/3366699

be extracted. Flow size refers to the number of elements (e.g., packets or bytes) in a flow. It has applications in heavy-hitter detection, flow distribution, super-changer detection, congestion diagnosis, routing loop and blackhole detection, etc. Flow spread refers to the number of *distinct* elements (e.g., header-field values or content) in a flow. It has applications in super-spreader detection, worm activity monitoring, DDoS attack detection, frequency profiling, etc. However, modern traffic measurement faces significant challenges below.

Per-flow measurement: This has been the norm with wide deployment of NetFlow [8] and its non-Cisco equivalent, which have well-known problems, including large space overhead [60], low sampling rate [16, 38] and flow size measurement only. There were successes in addressing some specific performance issues [18, 33], but the fundamental problem of NetFlow's high space overhead remains untackled. To circumvent this problem, most research has generally followed the path of reducing measurement scope by abandoning per-flow measurement and focusing on a small number of heavy hitters or statistics collection [1, 15–17, 29, 31, 34, 49]. However, while per-flow data will allow us to find heavy hitters, detect traffic changes, learn temporal-spatial traffic distributions, and compute statistics such as entropy, the other way around is not true — heavy hitters and statistics cannot substitute the value of per-flow traffic measurement, which is more generic and versatile in supporting network management functions, particularly when one wants to investigate the behavior of individual hosts or aggregate of selected subsets. Some applications of per-flow measurement are stealthy attack detection, fine-grained traffic analysis, flow loss map, ECMP debugging, and TCP timely attack detection [33].

Generality: There can be numerous different measurement tasks in a network. Implementing a separate hardware/software solution for each one is costly and inefficient. It is highly desirable to have a universal solution that fits many tasks and is future-proof in handling unforeseen tasks after deployment [34]. However, most existing work is limited to specific measurement requirements. First, many solutions (including NetFlow) focus on flow-size measurement and heavy hitters [3, 34, 58] but ignore flow-spread measurement. In their data structures, counters suffice in tracking the number of packets in a flow. But that is not adequate in counting the number of *distinct* elements in a flow, which requires a data structure that can "remember" the elements it has seen before in order to avoid counting duplicates of the same elements, where a flow may have millions of elements to "remember". It takes hundreds or thousands of bits per flow to do so [19, 22, 26, 54], which presents a serious challenge when there are tens of thousands or millions of concurrent flows. Second, the designs of current traffic monitoring systems are often geared toward specific performance goals such as space compactness [34, 35, 59] or estimation range [56]. They are not general enough for complex and flexible tradeoff among accuracy, throughput, space and processing overhead.

Low space/processing overhead: Modern routers forward packets from incoming ports to outgoing ports via switching fabric. To match the line speed, on-chip cache memory is often the first choice for online network functions. However, limited on-chip memory may have to be shared among routing, performance, measurement, security and other functions, each of which can only use a fraction of the available space. Depending on their relative importance, some functions may be allocated tiny portions of the available memory, whereas the amount of data they have to process and store can be extremely large in high-speed networks. The great disparity between memory supply and traffic volume requires us to implement traffic measurement functions as compactly as possible. As an example, if the amount of on-chip memory allocated to a measurement task is 0.5MB but there are 1M concurrent flows, with 4 bits per flow, can we still perform per-flow size or spread measurement, with

generality in performance tradeoff? In addition to space efficiency, we also want to keep per-packet processing time to a minimum constant, such as a single memory update per packet. Constant processing steps (time) for each packet helps avoid pipeline stalls and thus optimize throughput.

Our goal is to meet all the above challenges by designing a family of per-flow measurement solutions that share a common implementation structure, which can be augmented for different tradeoff requirements, measuring both flow size and flow spread with constant per-packet overhead (down to one memory update and a couple of hashes), and doing so in tight memory (down to several bits per flow). In addition, we want to support network-wide distributed measurement. No prior art has achieved such a goal. UnivMon [34] does not perform per-flow measurement, nor does it consider flow spread. It incurs variable processing time with numerous memory accesses per packet in the worst case. Elastic Sketch [58] does not measure flow spread and incurs variable processing overhead. SketchLearner [28] needs to update $l + 1$ sketches per packet, each requiring multiple memory accesses, with variable per-packet time, and it does not consider spread, where l is the number of bits in flow ID. The above work can estimate the sizes of flows, not the spreads of individual flows. None of them provides a generalized structure for accuracy/throughput/space/processing tradeoff. FlowRadar [33] provides a constant-time solution, but cannot fully address the space challenge faced by its NetFlow-type design, which is less efficient than the space-sharing sketches that we adopt. RHHH [3] monitors hierarchical heavy hitters with constant per-packet time (many memory accesses), but it does not consider flow spread or per-flow measurement. Our work is complementary to OpenSketch [61] and SketchVisor [27], and can be integrated into their architectures.

Numerous sketches have been proposed for traffic measurement [1, 5, 6, 9, 12–15, 19, 34, 39, 42, 62], each designed under specific performance requirements. This paper does not intend to propose yet another sketch as numerous existing ones fail in fulfilling the goal stated previously. Instead, we take a different path towards introducing meta-level frameworks that incorporate the seemingly independent past (and future) sketches under a common implementation structure, with the flexibility of plug-n-play and many options for tradeoff. With this idea in mind, we propose several families of measurement solutions, using the existing sketches as building components, and we compare them to find their respective areas of excellence in a multi-dimensional evaluation space of measurement accuracy, memory efficiency, processing overhead, online throughput and query performance. The contributions are summarized below.

First, we propose a family of sketches called bSketch based on the structure underlying the counting Bloom filter, and another family called cSketch based on the structure of CountMin. These sketches share the same implementation, with different plug-in data structures (e.g., counters, bitmaps [54], HLL [26] or a combination of them) for actual packet recording. They support measurement of flow size/spread or multiple measurement tasks together.

Second, we introduce a family of virtual sketches called vSketch, also using a plug-in design, which shares memory at a fine level, enabling per-flow measurement for both size and spread with one memory update per packet in tight memory averaging several bits per flow.

Third, we discuss how to apply these sketch families in distributed measurement.

Fourth, we implement the proposed sketch families in both hardware and software. We compare their performance using trace-based experiments and reveal the multi-dimensional tradeoff that these sketches represent. Our experimental results shed light on how to choose different sketches based on performance and application goals.

2 BACKGROUND

2.1 Flow Model

We adopt a generalized flow model where each flow consists of all packets that share the same value(s) in a pre-defined subset of header fields, which together form the flow ID. Then an incoming packet stream can be abstracted as a sequence of $\langle f, e \rangle$ pairs, where f is a flow ID and e is an element to be measured. With different flow ID definition, flows under measurement may be per-source flows, per-destination flows, per-source/destination flows, TCP flows, WWW flows, P2P flows, or other application-specific flows. Flow spread and flow size are two fundamental traffic measurements based on this flow model. For example, NetFlow contains several size measurement for TCP/UDP flows. Higher-order statistics such as mean, deviation, skewness and histogram among all or a selected set of flows can be derived from per-flow measurements [4, 34].

Flow spread is defined as the number of *distinct* elements in each flow, where elements under measurement may be destination addresses, source addresses, ports, or even application-header fields. Consider per-source flows, where all packets from the same source address constitute a flow. Measuring the number of distinct destination addresses in each flow not only helps detect scanning activities [48] but also provides information for monitoring the actual scanning rate, which can assist in determining the characteristics of malicious activities, for instance, the infection rate in outbreak of random-scanning worm attacks [7, 40, 50]. Consider per-destination flows, where all packets sent to a common destination constitute a flow. If the number of distinct source addresses in a flow suddenly surges, it may signal a DDoS attack [37, 41, 46, 47, 52] against the destination of the flow. We may even consider all HTTP request packets that carry the same URL as a flow. An institutional gateway may determine the popularity of external web content for caching priority by tracking the number of distinct source addresses in each flow (which consists of all outbound HTTP requests for the same web content).

Flow size is defined as the number of elements in each flow, where elements may be packets, bytes, or occurrences of a certain header-field value. One may measure the number of packets in each TCP flow, the data rate of each voice-over-IP session, the number of bytes that each host downloads, the number of SYN packets from each source address, or the number of ACK packets sent to each address. Such information is very useful to service provision, capacity planning, accounting and billing, and anomaly detection [25, 30]. For instance, measuring the number of SYN/ACK packets provides a means for detecting SYN attacks [53]. Studying per-flow sizes over consecutive measurement epochs helps network operators discover activity patterns and together with user profiling, reveal geographic/demographic traffic distributions among users. Such information assists Internet service providers and application developers to align network resource allocation with the majority's needs [32]. In the event of a botnet attack where there is a sudden surge of small flows, a security administrator may analyze the change in the flow size distribution [16, 29] and use per-flow information to compile the list of candidate bots that contribute to the change [23, 24].

Measuring flow spread is more difficult than measuring flow size. Consider per-source flows. If a source sends 10,000 packets to 100 different destinations, its flow size is 10,000 packets, but its flow spread is 100 distinct destinations. For flow spread, there needs a mechanism to keep track of duplicates in the flow, which takes a lot of space.

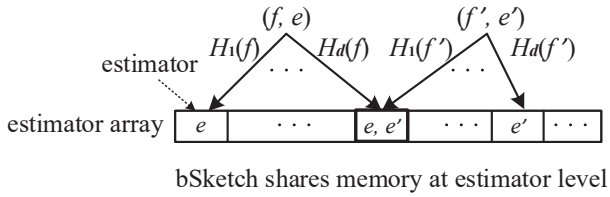


Fig. 1. Flow f is hashed to d estimators where its elements e are recorded. Two flows, f and f' may be hashed to the same estimator, which records the elements of both flows.

2.2 System Model

Traffic measurement functions are implemented by monitors and a controller. The monitors are software/hardware modules that are deployed on routers, switches, firewalls, or middleboxes to record per-flow statistics in real-time. Each monitor is implemented by an ID module and a measurement module. The ID module processes only selected packets (such as SYN packets) or samples a subset of packets to record flow IDs (such as source addresses for per-source flows). Sampling may lose some flow IDs, particularly for UDP, but it tends to keep those of larger flows, especially ones that persist over multiple measurement epochs. Because flow IDs are not recorded on a per-packet basis, this module may be implemented in off-chip memory. The bigger problem is the on-chip measurement module which processes all arrival packets. Each packet is abstracted as an $\langle f, e\rangle$ pair, and the measurement module records e in the data structure designated for f. The focus of this paper is the measurement module.

Time is divided into epochs. At the beginning of each epoch, the monitors send the measurement results to the controller and reset their data structures for the next epoch. The controller will combine the measurement results from the whole network and answer user queries.

3 BSKETCH: A FAMILY OF BLOOM SKETCHES

We begin by presenting our first plug-n-play sketch family called bSketch, which helps introduce the basic design approach that also underlies more sophisticated sketches later. Our discussion will surround two abstract tasks: size measurement and spread measurement. We do not refer to numerous application-specific measurement tasks because they can be abstracted into the above two.

3.1 Counting Bloom Filter

The counting Bloom filter [9] was originally designed to support member deletion in a Bloom filter. We can use it for flow-size measurement as follows: The data structure is a single array of l counters, denoted as C. For each packet $\langle f, e\rangle$, we hash f to d counters, and increases these counters by one, i.e., $C[H_i(f)] := C[H_i(f)] + 1$, where H_i, $0 \le i < d$, are independent hash functions whose output range is $[0, l)$ through modulo operation. Similar to CountMin [12] (which will be discussed shortly), when querying about flow f, we again hash f to the d counters and return the minimum value of them as the estimated flow size. The estimate is accurate when one of the d counters has no other flow hashed to it. But if all d counters are shared with other flows, there will be an estimation error. By choosing the value with the smallest error among the d counters, we can reduce the error very effectively.

3.2 bSketch Design

We generalize the structure of counting Bloom filter to a family of sketches, called *bSketch*, where the counting Bloom filter represents a point in this much larger solution space with much expanded functionalities. As illustrated in Fig. 1, the counter array is replaced with an *estimator array*, where each estimator is a plug-in that can be any data structure for measuring a certain quantity of a flow, which may be size, spread or other statistics. The technical contributions of bSketch are three-hold: First, its generalized structure expands the measurement scope from flow size to flow spread and other statistics or their combinations. Second, it provides a common framework of implementation with plug-ins for a family of member solutions, as is done in our hardware/software experiments. Third, we will thoroughly compare its member solutions together with other sketches experimentally to find their respective strengths and weaknesses under various application goals and resource settings.

The data structure of bSketch is an array of l estimators, denoted as A. Notations introduced here or later can be found in Table 1 for quick reference. There are two basic operations: recording an incoming packet $\langle f, e\rangle$ and querying the size/spread of a flow f. For recording, we hash f to d estimators in the array and record e in those estimators $A[H_i(f)]$, $0 \le i < d$, as shown in Alg. 1, where the hash functions may be practically implemented from a master hash function H as $H_i(f) = H(f \oplus s[i])$, and s is an array of m randomly selected seeds. The actual recording operation as a plug-in at Line 2 is dependent on the type of estimators in use; its pseudo code is given in Alg. 2 and will be discussed shortly. To query about flow f, we hash f to the d estimators and return the minimum value produced from these estimators, as shown in Alg. 3 with another plug-in at Line 4.

Algorithm 1 Recording a packet in bSketch

Input: estimator array A, seed array s, packet $\{\langle f, e\rangle\}$
Action: record e of f in A

1: **for** $i = 0..d-1$ **do**
2: $\quad bRecord\_Plugin(f, e, A[H_i(f)])$
3: **end for**

Algorithm 2 $bRecord\_Plugin(f, e, A[H_i(f)])$

Input: flow label f, element id e, estimator $A[H_i(f)]$
Action: record e of f in $A[H_i(f)]$

1: **switch** (type of estimator $A[H_i(f)]$)
2: **case** counter**:**
3: $\quad A[H_i(f)] := A[H_i(f)] + 1$
4: **case** bitmap**:**
5: $\quad A[H_i(f)][H(e)] := 1$
6: **case** FM**:**
7: $\quad$ set $G(e)$th bit in $A[H_i(f)][H(e)]$ to one
8: **case** HLL**:**
9: $\quad A[H_i(f)][H(e)] := \max\{A[H_i(f)][H(e)], G(e)\}$
10: **end switch**

f	flow label
e	element id
k	actual size/spread of flow f
$\hat{k}$	estimated size/spread of flow f
$H, H_i(\cdot)$	uniform hash function
s	array of randomly selected seeds
$G(\cdot)$	geometric hash function
l	total number of estimators in bSketch
d	number of estimators per flow in bSketch
$A[j]$, $0 \le j < l$	estimator array in bSketch
b	bitmap size in bSkt(bitmap)
m	number of estimator arrays in vSketch
w	size of each array in vSketch
$U_i[j]$, $0 \le i < m$, $0 \le j < w$	jth unit-sketch of ith array in vSketch
L_f	logical estimator for flow f
x	number of elements recorded in $A[H_i(f)]$, number of elements recorded in L_f
$\hat{x}$	estimated number of elements in L_f
n	noise in $A[H_i(f)]$, noise in L_f
$\hat{n}$	estimated noise in L_f
L_F	logical estimator for super flow F
X	total sizes/spreads of all flows
$\hat{X}$	estimated total sizes/spreads of all flows

Table 1. Notations

Algorithm 3 Querying on a flow in bSketch

Input: estimator array A, seed array s, flow label f, maximum integer value INT_MAX
Output: size/spread estimate $\hat{k}$

1: $\hat{k} := INT\_MAX$
2: **for** $i = 0..d-1$ **do**
3: $\quad \hat{k} := \min\{\hat{k}, bQuery\_Plugin(A[H_i(f)])\}$
4: **end for**
5: **return** $\hat{k}$

bSketch represents a family of sketches, one for a different estimator plug-in. For example, when we plug in a counter for each estimator, as shown in Line 3 of Alg. 2, we have the counting Bloom filter, also denoted as bSkt(counter) in the family. Similarly, when we plug a bitmap [54], FM [22] or HLL [26] for each estimator in A, we have bSkt(bitmap), bSkt(FM) or bSkt(HLL), which can measure the spreads of all flows simultaneously in a memory averaging several bits per flow in our experiments, thanks to estimator sharing in Fig. 1, whereas the original designs of bitmap/FM/HLL [22, 26, 54] without sharing will take hundreds or thousands of bits to measure the spread of a single flow. We stress that the members in the bSketch family are not limited to those described below as other data structures may also be plugged in.

Algorithm 4 *bQuery_Plugin*($A[H_i(f)]$)

Input: estimator $A[H_i(f)]$
Action: size/spread estimate $\hat{k}$

1: **switch** (type of estimator $A[H_i(f)]$)
2: **case** counter**:**
3: $\hat{k} := A[H_i(f)]$
4: **case** bitmap**:**
5: $Z = 0$
6: **for** $j = 0..b-1$ **do**
7: **if** $A[H_i(f)][j] = 0$ **then** $Z := Z + 1$
8: **end for**
9: $\hat{k} := -b \ln(\frac{Z}{b})$
10: **case** FM**:**
11: $p = 0$
12: **for** $j = 0..m-1$ **do**
13: $p := p+$ no. of consecutive leading ones in $A[H_i(f)][j]$
14: **end for**
15: $p := \frac{p}{m}$; $\hat{k} := m2^p/\varphi$
16: **case** HLL**:**
17: $\alpha_m := 0.7213/(1 + 1.079/m)$; $sum := 0$
18: **for** $j = 0..m-1$ **do**
19: $sum := sum + 2^{-A[H_i(f)][j]}$
20: **end for**
21: $\hat{k} := \alpha_m m^2/sum$
22: **end switch**
23: **return** $\hat{k}$

3.3 bSkt(bitmap)

Each estimator $A[j]$, $0 \le j < l$, is implemented as a bitmap of b bits, which are referred to as $A[j][u]$, $0 \le u < b$. The total memory size is $b \times l$. All bits are reset to zeros at the beginning of each measurement epoch.

Record: An arrival packet $\langle f, e \rangle$ is hashed to estimator $A[H_i(f)]$, $0 \le i < d$. To record element e, we hash e to a bit in the bitmap of $A[H_i(f)]$ and set the bit to one: $A[H_i(f)][H(e)] := 1$, as shown in Line 5 of Alg. 2, where H is another hash function. Regardless of how many times e appears in flow f, because they set the same bit, the duplicates are automatically filtered as they make no extra impact on the bitmap.

Query: When a query is made on flow f, for each bitmap $A[H_i(f)]$, we estimate the number of distinct elements recorded as $\hat{k}_i = -b \ln V_i$ from [54], as shown in Line 5-9 of Alg. 4, where V_i is the fraction of bits in $A[H_i(f)]$ that are zeros. Finally, we return $\min_{i \in [0,d)}\{\hat{k}_i\}$ as an estimation for flow f's spread. The estimation range is up to $b \ln b$.

3.4 bSkt(FM)

Each estimator $A[j]$, $0 \le j < l$, consists of m FM unit-sketches [22], which are referred to as $A[j][u]$, $0 \le u < m$. Each unit-sketch, $A[j][u]$, has 32 bits, which are initialized to zeros at the beginning of each epoch.

Record: To record element e in estimator $A[H_i(f)]$, we first hash e to one of its unit-sketches, $A[H_i(f)][H(e)]$, and then record e in that unit-sketch as follows: Perform a geometric hash $G(e)$ whose output is i with a probability of 2^{i+1} and set the $G(e)$th bit of $A[H_i(f)][H(e)]$ to one, as shown in Line 7 of Alg. 2.

Query: When a query is made on flow f, for each estimator $A[H_i(f)]$, $0 \leq i < d$, we estimate the number of distinct elements as $\hat{k}_i = m2^{z_i}/\varphi$, as shown in Line 11-15 of Alg. 4, where z_i is the average number of consecutive ones (starting from the least significant bit) in the unit-sketches of $A[H_i(f)]$, and φ is a bias correction constant whose value can be found in [22]. Finally, we return $\min_{i\in[0,d)}\{\hat{k}_i\}$ as an estimation for flow f's spread. The estimation range is up to 2^{32}.

3.5 bSkt(HLL)

Each estimator $A[j]$, $0 \leq j < l$, consists of m HLL unit-sketches, which are referred to as $A[j][u]$, $0 \leq u < m$. Each unit-sketch, $A[j][u]$, is 5 bits long, which is initialized to zero at the beginning of every measurement epoch.

Record: An arrival packet $\langle f, e\rangle$ is hashed to estimator $A[H_i(f)]$, $0 \leq i < d$. To record element e, we hash e to one of its unit-sketches, $A[H_i(f)][H(e)]$, where $H(e)$ is another hash function. We then record e in that sketch as follows: Perform a geometric hash $G(e)$ whose value is i with a probability of 2^{i+1}, and if $G(e) > A[H_i(f)][H(e)]$, replace the value of $A[H_i(f)][H(e)]$ with $G(e)$, as shown in Line 9 of Alg. 2. One way to implement $G(e)$ is to perform a uniform hash $H'(e)$ and return the number of leading zeros in the hash result.

Query: Consider estimator $A[H_i(f)]$, $0 \leq i < d$. We define the harmonic mean of its m unit-sketches as $har(A[H_i(f)]) = (\sum_{u=0}^{m-1} 2^{-A[H_i(f)][u]})^{-1}$. When a query is made on flow f, from each estimator $A[H_i(f)]$, we estimate the number of distinct elements as $\hat{k}_i = \alpha_m \cdot m^2 \cdot har(A[H_i(f)])$, as shown in Line 17-21 of Alg. 4, where α_m is a bias correction constant whose value can be found in [21]. According to [21], when the estimate is small, we need to convert the estimator $A[H_i(f)]$ into a bitmap for estimation, which is also true for FM [22]. Finally, we return $\min_{i\in[0,d)}\{\hat{k}_i\}$ as an estimation for flow f's spread. The estimation range is up to 2^{32}.

3.6 Estimation Accuracy

Below we derive the mean and variance of bSketch estimate $\hat{k}$ for an arbitrary flow f whose true size/spread is k. All k elements of flow f are recorded by each of its d estimators, $A[H_i(f)]$, $0 \leq i < d$, which also record a number n of noise elements from other flows due to estimator sharing. Let x be the total number of elements recorded by $A[H_i(f)]$. We have $x = k + n$.

For the estimate $\hat{k}_i$ from $A[H_i(f)]$, its mean and standard deviation, denoted as $\mu(n)$ and $\sigma(n)$, are dependent on the instance value of noise n. Their exact formulas depend on the estimator type and can be found in [21, 22, 54] for bitmap, FM and HLL, respectively. Because $\hat{k}_i$ is computed from the sum of a large number m of unit-sketches in $A[H_i(f)]$, by the central limit theorem, it follows approximately a normal distribution, $N(\mu(n), \sigma(n))$. The cumulative distribution function (CDF) of $\hat{k}_i$ with respect to n is given as $G(n,t) = \frac{1}{2}(1 + erf(\frac{t-\mu(n)}{\sqrt{2}\sigma(n)}))$, where $erf(.)$ is the complementary error function.

Let X be the total number of elements from all flows. $X - k$ is the number of elements from flows other than f. They are noise elements with respect to f, and each of them are recorded for d times by randomly chosen estimators. Assume that noise is randomly

distributed among all estimators in A. The number n of noise elements recorded by one estimator $A[H_i(f)]$ follows a Binomial distribution, $n \sim Biom(d(X-k), \frac{1}{l})$. Hence, for $0 \le j < d(X-k)$,

$$Prob\{n=j\} = \binom{d(X-k)}{j}(\frac{1}{l})^j(1-\frac{1}{l})^{d(X-k)-j}. \tag{1}$$

Considering the above noise distribution, the CDF of $\hat{k}_i$ is

$$\begin{aligned} G^*(t) &= \sum_{j=0}^{d(X-k)} Prob\{n=j\}G(j,t) \\ &= \sum_{j=0}^{d(X-k)} \binom{d(X-k)}{j}(\frac{1}{l})^j(1-\frac{1}{l})^{d(X-k)-j}G(j,t). \end{aligned} \tag{2}$$

Because $\hat{k} = \min_{i\in[0,d)}\{\hat{k}_i\}$, the CDF of $\hat{k}$, i.e., the probability of $\hat{k} \le t$, for any $t > 0$, is given as

$$F(t) = 1 - (1-G^*(t))^d. \tag{3}$$

The probability distribution function (PDF) of $\hat{k}$ can be calculated by differentiating $F(t)$ with respected to t,

$$\begin{aligned} f(t) &= d(1-G^*(t))^{d-1} \sum_{j=0}^{d(X-k)} Prob\{x=k+j\}g(j,t) \\ &= d(1-G^*(t))^{d-1} \sum_{j=0}^{d(X-k)} \binom{d(X-k)}{j}(\frac{1}{l})^j(1-\frac{1}{l})^{d(X-k)-j}g(j,t), \end{aligned} \tag{4}$$

where $g(j,t) = \frac{1}{\sqrt{2\pi\sigma^2(j)}}e^{-\frac{(t-\mu(j))^2}{2\sigma^2(j)}}$, which is the PDF of normal distribution. Therefore, the expectation and variance of bSketch, denoted as $\hat{\mu}$ and $\hat{\sigma}^2$, can be calculated as

$$\begin{aligned} \hat{\mu} &= E(\hat{k}) = \int_0^\infty tf(t)dt, \\ \hat{\sigma}^2 &= Var(\hat{k}) = \int_0^\infty (t-\hat{\mu})^2 f(t)dt. \end{aligned} \tag{5}$$

By the Chebyshev inequality [55], the bias, $\hat{k}-\hat{\mu}$, is bounded by $r\hat{\sigma}$ with probability

$$Prob(|\hat{k}-\hat{\mu}| \ge r\hat{\sigma}) \le \frac{1}{r^2}, \tag{6}$$

where r is an arbitrary positive real number.

3.7 Composite bSketch

We define a composite bSketch as one that performs multiple measurements together. For example, in bSkt(Counter, HLL), each estimator contains a counter for flow-size measurement and an array of m HLL unit-sketches for flow-spread measurement, allowing two measurements to be performed simultaneously in a single processing thread on the same data structure with shared hashing and memory access. Two more examples of composite bSketches are (1) two counters per estimator for counting the number of packets and the number of bytes in each flow, and (2) one bitmap and an array of m HLL unit-sketches per

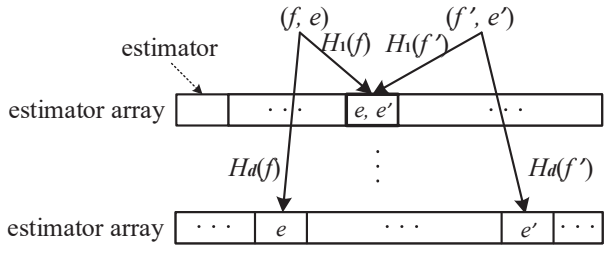


Fig. 2. Design of cSketch with multiple estimator arrays

estimator for counting two types of flow spread, one with a smaller required range and the other with a larger range.

3.8 CountMin and cSketch

CountMin [12] is similar to the counting Bloom filter (CountBlm) except for using d counter arrays to avoid intra-flow hash collision when mapping the same element to its d counters. CountMin can also be extended into a family of sketches, called *cSketch*. We describe it as follows. It consists of d estimator arrays, denoted as A_i, $0 \leq i < d$, each of which has w estimators. For each arrival packet $\langle f, e \rangle$, we hash f to one estimator in each array and record e in that estimator $A_i[H_i(f)]$, $0 \leq i < d$. When we want to query about flow f, we again hash f to one estimator in each array and return the minimum value among the measurement estimations produced by these d estimators.

An illustration of cSketch is shown in Fig. 2, where two flows f and f' share a common estimator in the first array, where both e and e' are recorded.

By implementing each estimator as a bitmap, FM, HLL or a combination of them, we derive cSkt(bitmap), cSkt(FM), and cSkt(HLL) or composite cSketch in a similar way as described in Section 3.

The relationship between CountMin and CountBlm is similar to that between partitioned Bloom filter and Bloom filter. Small flows have slightly smaller chance of colliding with large flows in CountBlm than in CountMin for the same reason that Bloom filter has slightly smaller false positive ratio than its partitioned counterpart [9]. That means bSketch has a slightly better performance than cSketch. But the difference is very small for small d values (e.g., 4) typically in use. Actually this type of design has been considered in [10, 11].

4 VSKETCH: A FAMILY OF VIRTUAL SKETCHES

bSketch requires multiple memory updates per packet. In this section, we design a family of virtual sketches, called *vSketch*, which incurs exactly one memory update per packet and works well in tight memory averaging several bits per flow. The contributions of vSketch are three-hold: First, its low overhead supports much higher packet throughput than bSketch, UnivMon [34] and Elastic Sketch [58] as our experiments will show. Second, it provides another common framework of implementation with plug-ins for a family of member sketches. Third, its generalized design embodies an effective mechanism for noise measurement and removal, making its member sketches more accurate than the prior work [56, 59] for multi-flow spread measurement.

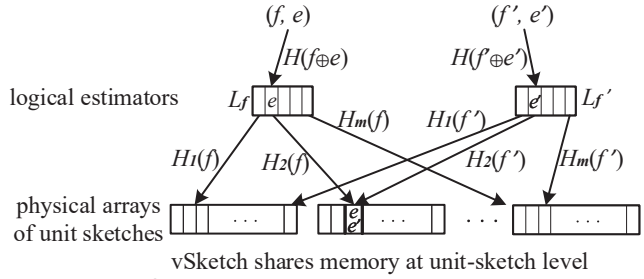


Fig. 3. Design of vSketch with unit-sketch sharing among logical estimators

4.1 Logical Estimators

The structure of vSketch is shown in Fig. 3, where each flow f is assigned to one *logical estimator* L_f, in contrast to bSketch that assigns each flow to d estimators. Consequently, vSketch records every incoming element once, compared to bSketch that does it d times.

Assigning a separate estimator to each flow is however costly in memory because an estimator for spread measurement takes hundreds or thousands of bits and there are numerous flows. Fortunately, our logical estimators are not real. They exist only in our mind. The physical data structure in vSketch consists of m arrays of unit-sketches, which are plug-ins that will be substituted with bits, counters, FM unit-sketches [22], HLL unit-sketches [26], or another data structure, each producing a different member in the vSketch family. The logical estimator of each flow is virtually constructed by randomly drawing (through hashing) one unit-sketch from each array, and it will record all elements of the flow in these m unit-sketches.

Two logical estimators may draw the same unit-sketch, resulting in memory sharing. vSketch shares memory among flows at the unit-sketch level, in contrast to bSketch that shares at the estimator level, which is evident by comparing Fig. 3 with Fig. 1. This finer level of sharing spreads elements from all flows more evenly across the memory space, resulting in better memory efficiency.

The benefit of sharing does not come for free. As shown in Fig. 3, e and e' from different flows are recorded in the same unit-sketch that is shared by f and f'. Because of such sharing, when any flow records its elements, it may introduce noise to the logical estimators of other flows. Fortunately, because unit-sketches are shared among flows uniformly at random, the noise is distributed among the unit-sketches randomly. vSketch is designed to measure and remove such noise effectively.

Algorithm 5 Recording a packet in vSketch

Input: unit-sketch arrays U, seed array s, packet $\{\langle f, e\rangle\}$
Action: record e of f in U

1: $i^* := H(f \oplus e)$
2: $vRecord\_Plugin(f, e, U_{i*}[H_{i^*}[f]])$

4.2 vSketch Design

Let U_i, $0 \le i < m$, denote the m arrays in vSketch, each of w unit-sketches. Consider an arbitrary flow f. Its logical estimator L_f consists of m unit-sketches, one from each array, i.e., $U_i[H_i(f)]$, $0 \le i < m$, where the hash functions may be practically implemented from a

Algorithm 6 $vRecord\_Plugin(f, e, U_{i*}[H_{i^*}[f]])$

Input: flow label f, element id e, unit-sketch $U_{i*}[H_{i^*}[f]]$
Action: record e of f in $U_{i*}[H_{i^*}[f]]$

```
1: switch (type of unit-sketch U_{i*}[H_{i*}[f]])
2: case counter:
3:    U_{i*}[H_{i*}[f]] := U_{i*}[H_{i*}[f]] + 1
4: case bit:
5:    U_{i*}[H_{i*}[f]] := 1
6: case FM unit-sketch:
7:    set the G(f ⊕ e)th bit in U_{i*}[H_{i*}[f]] to one
8: case HLL unit-sketch:
9:    U_{i*}[H_{i*}[f]] := max{U_{i*}[H_{i*}[f]], G(f ⊕ e)}
10: end switch
```

Algorithm 7 Querying on a flow in vSketch

Input: unit-sketch arrays U, seed array s, flow label f
Output: size/spread estimate $\hat{k}$

```
1: for i = 0..m − 1 do
2:    L_f[i] = U_i[H_i(j)]
3:    turn U_i into L_F[i]
4: end for
5: x̂ = vQuery_Plugin(L_f)
6: X̂ = vQuery_Plugin(L_F)
7: n̂ = X/w
8: return k̂ = x̂ − n̂
```

master hash function H as $H_i(f) = H(f \oplus s[i])$, and s is an array of m randomly selected seeds. That is, the ith unit-sketch in the logical estimator, denoted as $L_f[i]$, is actually $U_i[H_i(f)]$ from the physical data structure.

To record an arrival packet $\langle f, e\rangle$, as shown in Alg. 5, we compute $i^* = H(e)$ and records e in $U_{i^*}[H_{i*}(f)]$, which is an action $vRecord\_Plugin$ dependent on the type of the unit-sketch. To query about flow f, as shown in Alg. 7, we hash f along with m random seeds s to find its logical estimator, $L_f = \{U_i[H_i(f)] \mid 0 \le i < m\}$, which records both the elements from f (information) and some elements from other flows (noise) due to unit-sketch sharing. Let x be the total number of elements recorded in L_f, which is the sum of the flow size/spread k plus noise n, i.e.,

$$x = k + n. \tag{7}$$

We estimate the noise as follows: Let X be the sum of the sizes/spreads of all flows. The number of elements from flows other than f is $X - k$; these are noise to f. They are distributed among the $m \times w$ unit-sketches randomly. Because L_f contains m unit-sketches, the portion of noise in L_f has the following expected value

$$E(n) = \frac{X - k}{w} \approx \frac{X}{k}, \tag{8}$$

if $k \ll X$. Consider a super flow F that contains all X elements from all flows. In the next subsection, we will explain how to turn U into the super flow's logical estimator L_F (Line 3

of Alg. 7), which will provide an estimate $\hat{X}$. By substituting X with its estimate $\hat{X}$ and using the mean noise as the estimate for n, we have $\hat{n} = \frac{\hat{X}}{w}$. Finally, by subtracting the mean noise, we estimate the flow's size/spread as $\hat{k} = \hat{x} - \frac{\hat{X}}{w}$ (Line 7-8 of Alg. 7), where $\hat{x}$ is the estimation of x computed from L_f through *vQuery_Plugin* which depends on the type of unit-sketches. The pseudo code of *vQuery_Plugin*, Alg. 8, is similar to Alg. 4, and we place it in Appendix I.

While the formula for $\hat{k}$ removes the mean noise, it cannot remove the variance in noise distribution, which results in estimation errors and may even make $\hat{k}$ negative. Hence, when $\hat{k}$ turns out to be negative, we set it to the smallest size/spread, 1.

4.3 Members in vSketch Family

By substituting each unit-sketch in $U_i[j]$, $0 \le i < m$, $0 \le j < w$, with a bit, a counter, a 32-bit FM unit-sketch [22] and a HLL unit-sketch [26], we derive members in the vSketch family, called *virtual sketches* and denoted as vSkt(bitmap), vSkt(counter), vSkt(FM) and vSkt(HLL), respectively. We briefly describe vSkt(counter) and vSkt(HLL), while the other two can be derived similarly.

For vSkt(counter), when recording an element e of flow f, we compute $i^* = H(f \oplus e)$ and increase $U_{i^*}[H_{i^*}(f)]$ by one, as shown in Line 3 of Alg. 6. When a query is made about flow f, we estimate the flow size as $\hat{x} - \frac{\hat{X}}{w}$, where $\hat{x} = \sum_{i=0}^{m-1} U_i[H_i(f)]$, as shown in Line 3-6 of Alg. 8. To estimate the size $\hat{X}$ of the super flow, we combine $U_0, U_1, ..., U_{m-1}$ into a super estimator L_F, where $L_F[i] = \sum_{j=0}^{w-1} U_i[j],\ 0 \le i < m$ and $\hat{X} = \sum_{i=0}^{m-1} L_F[i]$.

For vSkt(HLL), when recording an element e of flow f, we compute $i^* = H(f \oplus e)$ and set $U_{i^*}[H_{i^*}(f)]$ to $G(f \oplus e)$ if the latter is greater than the former, as shown in Line 9 of Alg. 6. When a query is made about flow f, we estimate the flow spread as $\hat{x} - \frac{\hat{X}}{w}$, where $\hat{x}$ is the estimation produced from the logical HLL estimator L_f. To estimate the spread $\hat{X}$ of the super flow, we combine U into a super estimator L_F, where $L_F[i] = \max_{j \in [0, w-1]} U_i[j],\ 0 \le i < m$ and $\hat{X}$ is the estimation produced from the super estimator L_F using Line 20-24 of Alg. 8 with L_f replaced by L_F.

While we do not describe vSkt(FM) and vSkt(bitmap), their super estimators are constructed as follows: For vSkt(FM), $L_F[i]$ is the bitwise OR of $U_i[j]$, $j \in [0, w-1]$; for vSkt(bitmap), L_F is the concatenation of U_i, $0 \le i < m$.

We can construct composite vSketches by setting each unit-sketch to be a composite and thus allowing multiple measurement tasks to be performed simultaneously. For example, in vSkt(Counter, HLL), each unit-sketch $U_i[j]$, $0 \le i < m$, $0 \le j < w$, contains a counter $U_i[j].C$ for flow-size measurement and an HLL estimator $U_i[j].H$ for flow-spread measurement.

4.4 Estimation Accuracy

THEOREM 1. *The expectation of a flow's vSketch estimate $\hat{k}$ satisfies*

$$k(1 - \epsilon - \frac{1}{w}) \le E(\hat{k}) \le k(1 + \epsilon - \frac{1}{w}), \tag{9}$$

where $\epsilon = \delta(1 - \frac{1}{w} + \frac{2X}{wk})$, if the type of logical estimators chosen in the vSketch ensures that when a number y of elements are recorded in an estimator, the expectation of its estimate $\hat{y}$ satisfies

$$y(1 - \delta) \le E(\hat{y}) \le y(1 + \delta), \tag{10}$$

where $\delta \ge 0$ and is dependent on the estimator type.

The proof can be found in Appendix II. The number w of unit-sketches in each array of vSketch is expected to be very large and thus $\frac{1}{w}$ is very small. If we ignore this small term in (9), ϵ specifies a bound on the estimation bias, i.e., $|E(\frac{\hat{k}}{k}) - 1| \leq \epsilon$. Its value is proportional to the bias δ of the estimator type in use. As is expected, it also increases as X (noise) increases, and it decreases as k (information) and w (memory space) increase. The value of δ is typically small. For example, when we use a counter estimator, $\delta = 0$. When we use an HLL estimator [21], $\delta = \epsilon_1(m)$, where $\epsilon_1(m) \leq 5 \times 10^{-5}$ if $m \geq 16$, and m is the number of unit-sketches in the HLL estimator. When we use an FM estimator [22], $\delta = \epsilon_2(m)$, where $\epsilon_2(m) \sim \frac{\lambda}{2m}$ if m is large, and λ is a constant which can be closely approximated as 0.61.

Theorem 2. *Assume that noise elements are distributed among all unit-sketches uniformly at random. The variance of vSketch estimate is*

$$Var(\hat{k}) = \begin{cases} \frac{X}{w}(1-\frac{1}{w}), & vSkt(counter) \\ m\alpha + \frac{\alpha\beta}{2w} + \frac{m\beta}{w}, & vSkt(bitmap) \\ \frac{0.78^2}{m}(k+\frac{X}{w})^2 + \frac{0.78^2}{mw^2}X^2 + (\frac{0.78^2}{m}+1)\frac{X}{m}(1-\frac{1}{w}), & vSkt(FM) \\ \frac{1.04^2}{m}(k+\frac{X}{w})^2 + \frac{1.04^2}{mw^2}X^2 + (\frac{1.04^2}{m}+1)\frac{X}{m}(1-\frac{1}{w}) & vSkt(HLL) \end{cases} \tag{11}$$

where $\alpha = e^{\frac{X}{mw}+\frac{k}{m}} - \frac{k}{m} - 1$ *and* $\beta = e^{\frac{X}{mw}} - \frac{X}{mw} - 1$.

The proof can be found in Appendix III. It is not always true that noise elements are distributed among all unit-sketches uniformly at random. A simple example is that when there are only two flows, the noise will not be distributed among all unit-sketches. However, this assumption is approximately true when there are a large number of flows and the size/spread of any flow is negligible when comparing with the total size/spread of all flows, which is generally true for large network traffic where sketches are needed.

5 DISTRIBUTED MEASUREMENT

Packets from a TCP flow tend to follow the same path. But our flow model includes both TCP flows and other generalized flows which may follow multiple paths. In this section, we show that bSketch and vSketch can support distributed measurement through simple spatial-temporal join operations. While other work such as [58] can also support distributed measurement, they do not have the generality in size/spread measurement, plug-n-play and multi-dimensional tradeoff that are sought after in this paper.

5.1 Spatial-Temporal Join for bSketch

We first consider spatial join. When a flow passes multiple paths, each router may only capture a fraction of the flow's traffic. For example, if we define a flow as all packets from the same source (which may be a single address or a subnet), then the flow may follow multiple paths when the source sends packets to different destination addresses. After routers send their bSketches to the controller, we need a primitive function called *spatial join* to combine these bSketches into one so that we can efficiently answer queries about the network-wide size/spread of a flow. Let n be the number of bSketches to be combined, and A^t be the array of estimators reported from the tth router, $0 \leq t < n$. Spatial join combines them into A^*. As is described below, the operation of spatial join is dependent on which sketch is used.

bSkt(counter): $A^*[j] = \sum_{t=0}^{n-1} A^t[j]$, for $0 \leq j < l$.

bSkt(bitmap): $A^*[j]$ is the bitwise OR of $A^t[j]$, $0 \leq t < n$, for $0 \leq j < l$.

bSkt(FM): The qth FM sketch in $A^*[j]$ is the bitwise OR of the qth sketches of $A^t[j]$, $0 \leq t < n$, for $0 \leq q < m$, $0 \leq j < t$.

bSkt(HLL): The qth HLL sketch in $A^*[j]$ is the maximum value among the qth sketches of $A^t[j]$, $0 \le t < n$, for $0 \le q < m$, $0 \le j < l$.

The query on the combined bSketch, A^*, is performed in the same way as presented in Section 3.

Next we consider temporal join. Each router may report its bSketch at a pre-defined schedule or when polled by a central controller. After reporting, it resets its bSketch to release space for recording new elements in the next epoch. If an application wants to know the flow size/spread over a period that covers multiple epochs, the controller may join the multiple reported bSketches from a router into one to support efficient queries. While the join is performed in temporal dimension, its operation is the same as that of spatial join.

5.2 Spatial-Temporal Join for vSketch

Consider the spatial join of n vSketches. Let U_i^t, $0 \le i < m$, be the vSketch from the tth router, $0 \le t < n$. Let U_i^*, $0 \le i < m$, be the resulting vSketch. The operation of spatial join is presented below.

vSkt(counter): $U_i^*[j] = \sum_{t=0}^{n-1} U_i^t[j]$, for $0 \le i < m$, $0 \le j < w$.

vSkt(bitmap): $U_i^*[j]$ is the OR of $U_i^t[j]$, $0 \le t < n$, $0 \le i < m$, $0 \le j < w$.

vSkt(FM): $U_i^*[j]$ is the bitwise OR of $U_i^t[j]$, $0 \le t < n$, for $0 \le i < m$, $0 \le j < w$.

vSkt(HLL): $U_i^*[j]$ is the maximum value among $U_i^t[j]$, $0 \le t < n$, for $0 \le i < m$, $0 \le j < w$.

The temporal join operation of n vSketches from the same router is the same as above.

6 EVALUATION

We evaluate the performance of the proposed sketch families in both hardware and software through trace-driven experiments to find tradeoff. We also compare them with the prior art, and discuss which sketches are more suitable for given performance goals.

6.1 Implementation

We implement bSketch, cSketch and vSketch in both hardware and software, including four sketches from each family: bSkt(counter), bSkt(bitmap), bSkt(FM), and bSkt(HLL) for bSketch; vSkt(counter), vSkt(bitmap), vSkt(FM) and vSkt(HLL) for vSketch. The performance of cSketch is very similar to that of bSketch. Therefore, our comparison will focus on bSketch vs. vSketch. We also implement the most relevant prior art, including Unimon [34], Elastic Sketch [58], CSE [59], vPCSA[57] and vHLL [56] for comparison. All related codes are available on Github [43].

OVS Implementation: They are implemented on OVS (OpenvSwitch) [45] in the kernel mode datapath under the Ubuntu 18.04LTS environment. We use the virtual network adapter on our machine to perform the experiments.

CPU Implementation: The sketch families are implemented on a machine with Intel Core i7-8700 3.2GHz CPU and 16GB memory.

GPU Implementation: We use the CUDA toolkit [44] for parallel programming of the proposed sketch families on GPU. Experiments are performed on an NVDIA GPU with GeForce GTX 1070, 8GB GDDR5 memory and 1920 CUDA cores at a clock rate of 1506-1683 MHz.

FPGA Implementation: We implement the sketch families on XLINX ZYBO-7010, with 512MB DDR3 DRAM, 240 KB Block RAM, and a clock rate of 50 MHz.

6.2 Experimental Setting

Traffic Traces: We download twenty anonymized traces from CAIDA [51], with each trace containing one minute network traffic, which has 18M∼20M packets. The average packet size in each trace is around 800 bytes. In our experiments, we set each epoch to one minute long.

For flow size, we define a flow as all packets from the same source address to the same destination address, and we measure the number of packets in each flow, which has application in building a traffic map. There are around 438K flows in a single epoch. For flow spread, we define a flow as all packets to the same destination, and we measure the number of distinct source addresses in each flow, which has application in DDoS detection. There are around 213K flows in a single epoch.

Parameter Setting: The total memory for a traffic measurement task is 1MB. Counters are 32 bits long. Each FM unit-sketch is 32 bits, and each HLL unit-sketch is 5 bits. The number m of unit-sketches in each FM or HLL estimator is 128. The number d of estimators that each flow is hashed to in bSketch is 4. The default parameter values may change in experiments in order to evaluate the impact of individual parameters. For example, we will reduce the memory to 0.25MB to evaluate how these sketches perform with several bits per flow on average.

bSkt(counter) and vSkt(counter) are designed for flow-size measurement. Other sketches, including bSkt(bitmap), bSkt(FM) and bSkt(HLL), vSkt(bitmap), vSkt(FM) and vSkt(HLL) are designed for spread measurement. But they can also be adapted for size measurement. For bSkt(bitmap) and vSkt(bitmap), instead of setting the $H(e)$th bit in the (logical) estimator, we simply take a random number r and set the rth bit in the (logical) estimator. Similarly for bSkt(FM), bSkt(HLL), vSkt(FM) and vSkt(HLL), we replace geometric hash $G(e)$ with a geometrically-distributed random number for size measurement.

The maximum flow size in each epoch is around 150K. Hence, we set the bitmap size in bSkt(bitmap) and vSkt(bitmap) to 20000 bits in order to accommodate the maximum size. The maximum flow spread in each epoch is around 20K. Hence, we set the bitmap size in bSkt(bitmap) and vSkt(bitmap) to 5000 bits in order to accommodate the maximum spread.

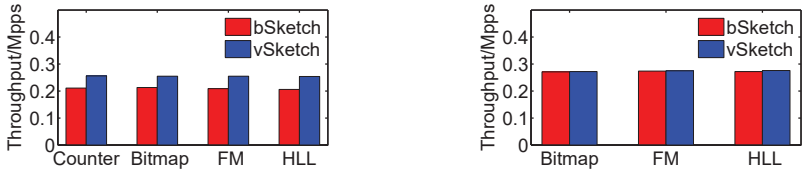
Performance Metrics: We compare sketches under different parameter settings in terms of online throughput, query speed, and estimation accuracy.

1. Online throughput. The average number of packets processed by the measurement module per second. Because the average packet size in our data set is 803 bytes. We can compute throughput in bits per second by multiplying the packet count with 6,424 bits per packet.

2. Query speed. The average number of queries that the controller (using the same machine as described previously) can process each second.

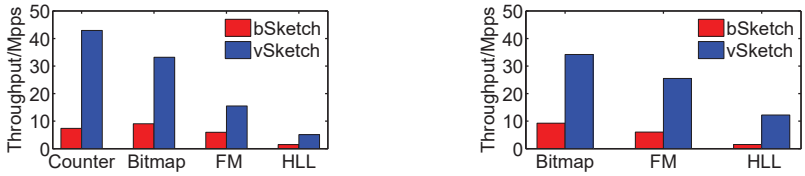
3. Estimation accuracy. We use both absolute error and relative error to measure the estimation accuracy. Let $\hat{k}_f$ be the size/spread estimate of a flow f, and k_f be the true size/spread. The absolute error is defined as $\frac{1}{N}\sum_f |\hat{k}_f - k_f|$, where N is the total number of flows. The relative standard error is $\sqrt{\frac{Var(\hat{k_f})}{k_f}}$.

Below we first perform comparative studies on the proposed sketch families, then compare them with the prior art, and finally give a summary on when to use which sketch.



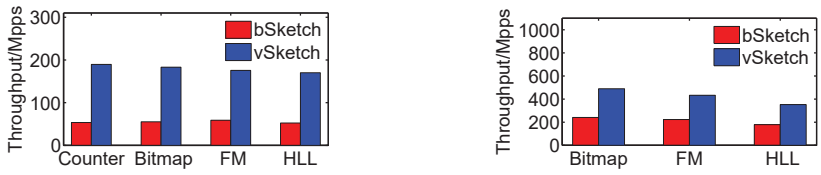
(a) Flow size (b) Flow spread

Fig. 4. Throughput comparison under OVS



(a) Flow size (b) Flow spread

Fig. 5. Throughput comparison without OVS



(a) Flow size (b) Flow spread

Fig. 6. Throughput comparison on GPU

(a) Flow size (b) Flow spread

Fig. 7. Throughput comparison on FPGA

6.3 Throughput

We first compare the throughput of our sketch family under different evaluation platforms including OVS, CPU, GPU and FPGA.

Fig. 4 compares online throughput of the eight sketches for measuring flow size and the six sketches for measuring flow spread in the two families, bSketch and vSketch. The sketches run in the dataplane of OpenvSwitch. All of them process the incoming packets with similar throughput of about 0.24 Mpps (mega packets per second), which is also the throughput when we turn off traffic measurement and run OVS alone. It suggests that these sketches contribute minor overhead in the dataplane of OVS and the throughput is largely determined by other switch operations.

When we run the sketches stand-alone on the same machine outside OVS, the throughput becomes much higher in Fig. 5 and differs among the sketches. As expected, vSketch has higher throughput than bSketch in general because of fewer memory updates and fewer hash

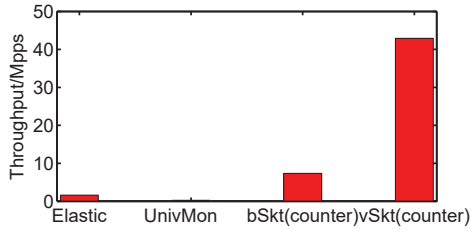
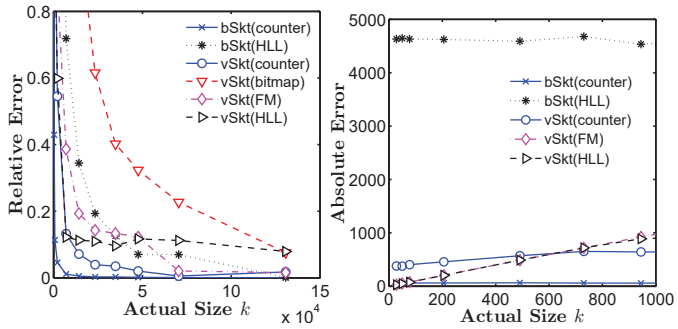


Fig. 8. Throughput comparison with prior work



(a) Relative error (b) Absolute error

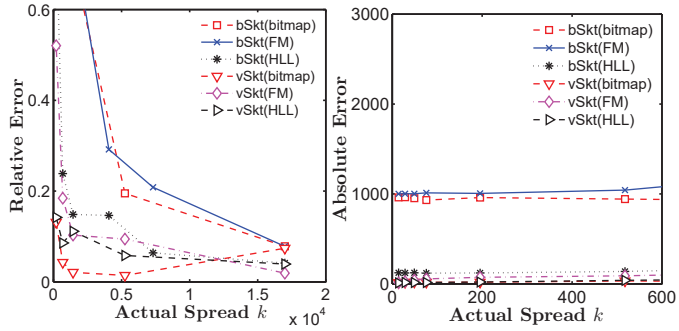
Fig. 9. Flow-size estimation accuracy

operations per packet. For size measurement, vSkt(counter) has the highest throughput of 42.93 Mpps (or 275 Gbps), compared to bSkt(counter)'s 7.35 Mpps. For spread measurement, the throughput ranges from 33.67 Mpps of vSkt(bitmap) to 12.2 Mpps of vSkt(HLL), compared to bSkt(bitmap)'s 9.07 Mpps and bSkt(HLL)'s 1.39 Mpps.

Throughput can be significantly increased with hardware accelerator such as GPU that offers parallel processing. Fig. 6 presents online throughput under GPU implementation. The highest throughput for size measurement is 190 Mpps by vSkt(counter) and the highest for spread measurement is 490 Mpps by vSkt(bitmap), which are 4.43 times and 14.99 times faster than the CPU implementation in Fig. 5, respectively. On one hand, hardware acceleration reduces throughput difference caused by processing. On the other hand, all sketches for size measurement need to generate a random number when recording an element. It appears that the random number generator cannot be parallelled, causing the throughput of size measurement to be smaller than that of spread measurement.

Fig. 7 presents online throughput under FPGA implementation. The vSketch members perform similarly, and so do the bSketch members, with the former's throughput four times of the latter. The reason is that bSketch members need to access memory more than vSketch members. vSketch makes one memory update per packet, compared to four updates by bSketch, resulting in 4:1 throughput ratio.

Next we compare with the prior art including Univmon [34] and Elastic Sketch [58]. Univmon also has a generic design, but it collects heavy hitters and traffic statistics, without per-flow measurement nor flow spread. Elastic Sketch combines a hash table for heavy hitters and CountMin for other flows. It is also designed only for flow size. Our relationship with Elastic Sketch is complementary because bSketch and vSketch can replace its CountMin part. Also closely related is FlowRadar [33], but it cannot be implemented under our experimental



(a) Relative error (b) Absolute error

Fig. 10. Flow-spread estimation accuracy

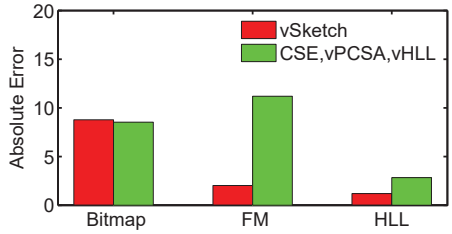


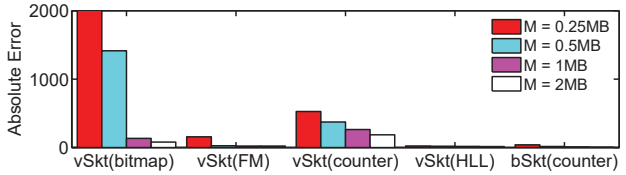
Fig. 11. Accuracy comparison with prior work

setting due to insufficient memory. Fig. 8 presents throughput comparison with the prior art for flow-size measurement on the CPU implementation under our default experiment setting described earlier. The throughput of UnivMon is 0.21 Mpps due to large per-packet overhead. Elastic Sketch achieves 1.16 Mpps for its two-stage processing; it has variable per-packet time. The throughput of bSkt(counter) and vSkt(counter) are much higher at 6.1 and 18.9 Mpps, respectively.

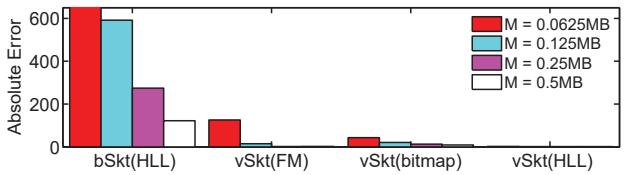
6.4 Accuracy and Memory Overhead

Fig. 9 compares bSkt(counter), bSkt(HLL), vSkt(counter), vSkt(bitmap), vSkt(FM) and vSkt(HLL) on flow-size estimation accuracy. We do not include bSkt(bitmap) and bSkt(FM) because their performance is much worse. Fig. 9a shows the relative errors. bSkt(counter) is clearly the winner, with vSkt(counter) in the second place for large flows. This is not surprising because these two sketches are specifically designed for flow-size measurement. Fig. 9a shows the absolute errors for small flows. bSkt(counter) again performs the best, vSkt(HLL) and vSkt(bitmap) come next, while bSkt(HLL) has large absolute errors for small flows.

Fig. 10 compares bSkt(bitmap), bSkt(FM), bSkt(HLL), vSkt(bitmap), vSkt(FM) and vSkt(HLL) on flow-spread estimation accuracy. Fig. 10a shows that vSkt(bitmap) performs the best with a small relative error of less than 4% for large flows, which is followed by 5% of vSkt(FM). The best in the bSketch family is bSkt(FM)'s 6%. It shows that noise removal of vSketches is more powerful than the min-value approach of bSketches for spread measurement. Fig. 10b shows absolute errors for small flows. Again vSkt(bitmap) performs the best, while the absolute errors of bSkt(bitmap) and bSkt(FM) are large for small flows.



(a) Flow-size estimation



(b) Flow-spread estimation

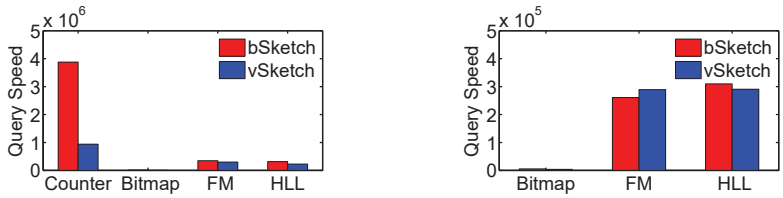
Fig. 12. Absolute error w.r.t. memory

In Fig. 12, we study estimation accuracy with respect to memory availability. Fig. 12a presents average absolute errors over all flows when the memory is 0.25MB, 0.5MB, 1MB and 2MB for size measurement and 0.0625MB, 0.125MB, 0.25MB, 0.5MB for spread measurement. It compares vSkt(bitmap), vSkt(FM), vSkt(counter), vSkt(HLL) and bSkt(counter) on flow-size measurement. As we increase memory, all sketches perform better. With 0.25MB, vSkt(bitmap) has the worst error, followed by vSkt(FM), vSkt(counter), and vSkt(HLL), while bSkt(counter) performs slightly worse than vSkt(HLL) under such tight memory. Fig. 12b compares bSkt(HLL), vSkt(FM), vSkt(bitmap), and vSkt(HLL) on flow-spread measurement. As we increase memory, all sketches perform better. In consistent with Fig. 10, vSkt(bitmap) and vSkt(HLL) have the smallest errors, with vSkt(HLL) slightly better, which works very well under 0.25MB, averaging serval bits per flow.

We also compare with the prior art. Elastic Sketch [58] uses CountMin, which is similar to bSkt(counter) for per-flow size measurement. On per-flow spread measurement, we compare CSE [59] (which uses bitmap) with vSkt(bitmap), vPCSA [57] with vSkt(FM), and vHLL [56] with vSkt(HLL). Fig. 11 shows the average absolute errors: vSkt(bitmap)'s 8.78 vs. CSE's 8.54, vSkt(FM)'s 2.03 vs. vPCSA's 11.20, 82% error reduction, and vSkt(HLL)'s 1.20 vs. vHLL's 2.84, 58% error reduction. While the two bitmap sketches perform similarly, vSketch outperforms vPCSA and vHLL significantly due to better noise estimation thanks to its ability to create an effective logical estimator for the super flow.

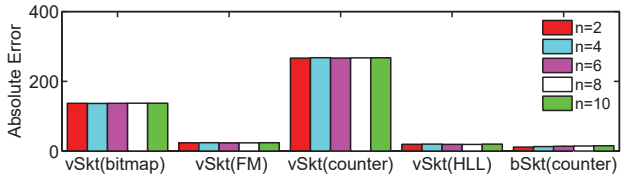
6.5 Query Speed

Fig. 13a shows that bSketch outperforms vSketch in offline query speed for flow size, with bSkt(counter)'s speed at 3.9 Mqps, compared to vSkt(counter)'s 0.9 Mqps, where Mqps stands for mega queries per second. The reason is that vSkt(counter) uses more counters in size computation than bSkt(counter). Two sketches, bSkt(bitmap) and vSkt(bitmap), have very low query speeds, 5.8 Kqps and 4.0Kqps respectively, because it takes time to scan their large bitmaps. Fig. 13b shows that bSketch and vSketch have more comparable performance for querying flow spread. Four sketches — bSkt(FM), bSkt(HLL), vSkt(FM) and vSkt(HLL) — achieve query throughput in the range of 2.7 to 3.1 Mqps.

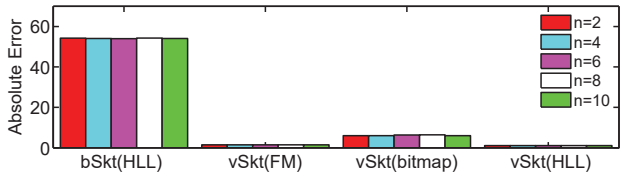


(a) Flow size (b) Flow spread

Fig. 13. Query speed comparison

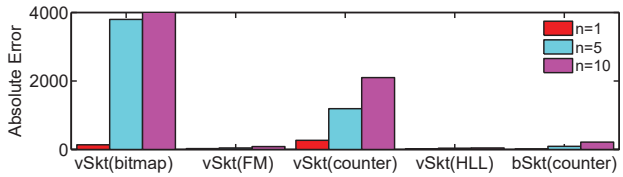


(a) Flow-size estimation

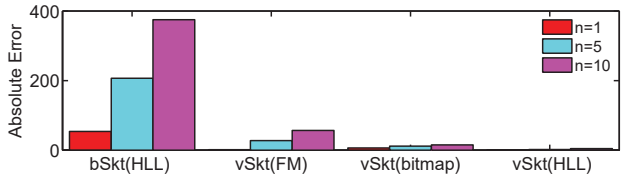


(b) Flow-spread estimation

Fig. 14. Aggregate error after spatial join



(a) Flow-size estimation



(b) Flow-spread estimation

Fig. 15. Aggregate error after temporal join

6.6 Spatial/Temporal Join

For network-wide distributed measurement, we use the traffic data from CAIDA and perform spatial/temporal join over multiple one-minute traffic traces we have downloaded. Fig. 14 compares the best five, vSkt(bitmap), vSkt(FM), vSkt(counter), vSkt(HLL) and

bSkt(counter), on average absolute errors over all flows after spatial join is performed on sketches received from $n = 2, 4, 6, 8, 10$ routers. The traffic traces from each router is one-minute long. The errors are largely independent of n, meaning that the sketches can all properly aggregate n parts into one for query. Fig. 14a shows that bSkt(counter) is clearly the winner for flow-size measurement and Fig. 14b shows that vSkt(HLL) is the winner for flow-spread measurement.

Fig. 15 compares the best four, bSkt(HLL), vSkt(FM), vSkt(bitmap) and vSkt(HLL), on average absolute errors over all flows after temporal join is performed on sketches received from one router in $n = 1, 5, 10$ epoches, during which traces at different times from the CAIDA download are fed to the same router. For all sketches, their accuracy drop as the number of epoches increases because more flows and more elements are recorded over time, introducing more noise in the data structures. Fig. 15a shows that bSkt(counter) is the winner for flow-size measurement. vSkt(bitmap) cannot handle $n = 10$ epoches because its limited estimation range is over-flown by too many flows and elements. Fig. 15b shows that vSkt(HLL) is the winner for flow-spread measurement.

6.7 Summary and Practical Guidelines

Based on our experimental comparison, we have the following conclusions: In terms of online throughput of stand-alone software implementation, vSkt(counter) performs the best for size estimation and vSkt(bitmap) performs the best for spread estimation. However, the difference among different sketches is largely diminished on hardware platforms or within OVS. In terms of query speed, bSkt(counter) performs the best for size estimation, while bSkt(HLL) performs the best for spread estimation. In terms of accuracy, bSkt(counter) performs the best for size estimation, while vSkt(bitmap) and vSkt(HLL) perform the best for spread estimation. The performance of all sketches is improved as we increase memory. bSkt(counter), vSkt(counter), vSkt(FM), vSkt(HLL), vSkt(bitmap) and bSkt(HLL) can properly handle spatial-temporal join.

For balance between online throughput and query speed, in stand-alone software and hardware implementations, one may choose vSkt(counter) or bSkt(counter) for size estimation; but in OVS, bSkt(counter) is the choice. Again for balance between the two, in stand-alone software and GPU implementation, one should choose vSkt(FM) for spread estimation; but in OVS and FPGA, one may choose vSkt(FM) or vSkt(HLL), according to Fig. 4-13. For balance between online throughput and estimation accuracy, one may choose vSkt(counter) for size estimation, and may alternatively choose bSkt(counter) in OVS implementation; one may choose vSkt(bitmap) or vSkt(FM) for spread estimation, and may alternatively choose vSkt(HLL) in hardware and OVS implementations, based on Fig. 4-7 and 9-12.

7 RELATED WORK

Much research on network traffic measurement has focused on heavy hitters and statistics [1, 5, 6, 13–15, 19, 34, 39, 42, 62]. Flow size measurement is a simple counting problem and can be easily solved using counters. However, when there are numerous flows, it can incur tremendous memory overhead. Therefore, most research has followed the path of designing efficient data structures to perform approximate estimation. Instead of using separate counters for individual flows, counting Bloom filter [9], CounterMin [12] and Counter Braids [35, 36] share counters among flows to reduce the memory overhead. To record one element, they have to update d counters, which limits their throughputs.

Flow spread measurement is a more difficult problem, and it is too costly to store all elements of each flow in a hash table for duplicate removal. Better solutions such as PCSA

[22], MultiresolutionBitmap [20], MinCount [2], LogLog [17] and HyperLogLog [21] provide efficient means to eliminate duplicates. But their memory consumption remains high. CSE [59] and vHLL [56] reduces memory consumption through space sharing.

UnivMon [34] introduces a multi-level sketch design that can measure many flow statistics at once. Monitoring hierarchical heavy hitters is a difficult problem, and Basat et al. proposes a randomized solution with constant per-packet processing time [3]. OpenSketch [61] provides a three-staged framework to compose traffic measurement functions from a component library. FlowRadar [33] provides a compact and constant-time design for NetFlow-type measurement. SketchVisor [27] proposes a software traffic measurement architecture with a fast path to handle surge in incoming traffic beyond the normal slow processing path can handle. Elastic Sketch [58] and SketchLearner [28] adopt a multi-staged design to isolate heavy hitters from the rest.

8 CONCLUSION

We presented three sketch families, called bSketch, cSketch and vSketch, each sharing a common implementation structure, based on which we can derive new sketches by plugging in different data structures that meet certain performance goal, application need or desirable tradeoff. We have implemented a number of sketches from each family and performed trace-based experiments to evaluate and compare their performance, which leads to guidance on how to pick from these sketches in practice.

Our future work is to extend research on network-wide measurement by exploring other sophisticated functions such as estimating the persistent traffic [63] in terms of both flow size and spread under the context of the proposed sketch families.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CNS-1719222 and STC-1562485 and a grant form Cyber Florida.

REFERENCES

[1] N. Bandi, D. Agrawal, and A. Abbadi. 2007. Fast Algorithms for Heavy Distinct Hitters using Associative Memories. *Proc. of IEEE International Conference on Distributed Computing Systems(ICDCS)* (June 2007).

[2] Z. Bar-yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. 2002. Counting Distinct Elements in a Data Stream. *Proc. of RANDOM: Workshop on Randomization and Approximation* (2002).

[3] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. *Proc. of ACM SIGCOMM* (2017).

[4] V. Braverman and R. Ostrovsky. 2010. Zero-one frequency laws. *in Proc. of STOC* (2010).

[5] J. Cao, Y. Jin, A. Chen, T. Bu, and Z. Zhang. 2009. Identifying High Cardinality Internet Hosts. *Proc. of IEEE INFOCOM* (April 2009).

[6] M. Charikar, K. Chen, and M. Farach-Colton. 2002. Finding Frequent Items in Data Streams. *Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)* (July 2002).

[7] S. Chen and Y. Tang. 2004. Slowing Down Internet Worms . *Proc. of IEEE ICDCS* (March 2004).

[8] Cisco. Online. Cisco IOS NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[9] S. Cohen and Y. Matias. 2003. Spectral Bloom Filters. *Proc. of ACM SIGMOD* (June 2003).

[10] J. Considine, M. Hadjieleftheriou, F. Li, J. Byers, and G. Kollios. 2009. Robust approximate aggregation in sensor data management systems. *ACM Transactions on Database Systems (TODS)* 34, 1 (2009), 6.

[11] G. Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers* (2011).

[12] G. Cormode and S. Muthukrishnan. 2004. An Improved Data Stream Summary: the Count-Min Sketch and Its Applications. *Proc. of LATIN* (2004).

[13] G. Cormode and S. Muthukrishnan. 2005. Space Efficient Mining of Multigraph Streams. *Proc. of ACM PODS* (June 2005).
[14] E. Demaine, A. Lopez-Ortiz, and J. Ian-Munro. 2002. Frequency Estimation of Internet Pacet Streams with Limited Space. *Proc. of Annual European Symposium on Algorithms (ESA)* (September 2002).
[15] X. Dimitropoulos, P. Hurley, and A. Kind. 2008. Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters. *ACM SIGCOMM Computer Communication Review* 38, 1 (2008), 7–16.
[16] N. Duffield, C. Lund, and M. Thorup. 2003. Estimating Flow Distributions from Sampled Flow Statistics. *Proc. of ACM SIGCOMM* (October 2003).
[17] M. Durand and P. Flajolet. 2003. Loglog Counting of Large Cardinalities. *ESA: European Symposia on Algorithms* (2003), 605–617.
[18] C. Estan, K. Keys, D. Moore, and G. Varghese. 2004. Building a better netflow. *Proc. of ACM SIGCOMM* (2004).
[19] C. Estan and G. Varghese. 2002. New Directions in Traffic Measurement and Accounting. *Proc. of ACM SIGCOMM* (August 2002).
[20] C. Estan, G. Varghese, and M. Fish. 2006. Bitmap Algorithms for Counting Active Flows on High-Speed Links. *IEEE/ACM Trans. on Networking* 14, 5 (October 2006).
[21] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. 2007. HyperLogLog: The Analysis of a Near-optimal Cardinality Estimation Algorithm. *Proc. of AOFA* (2007), 127–146.
[22] P. Flajolet and G. N. Martin. 1985. Probabilistic Counting Algorithms for Database Applications. *J. Comput. System Sci.* 31 (September 1985), 182–209. Issue 2.
[23] G. Gu, R. Perdisci, J. Zhang, and W. Lee. 2008. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-independent Botnet Detection. *Proc. of ACM Conference on Security Symposium* (2008).
[24] G. Gu, J. Zhang, and W. Lee. 2008. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. *Proc. of Network and Distributed System Security Symposium* (2008).
[25] F. Hao, M. Kodialam, and T. V. Lakshman. 2004. ACCEL-RATE: A Faster Mechanism for Memory Efficient Per-flow Traffic Estimation. *Proc. of ACM SIGMETRICS/Performance* (June 2004).
[26] S. Heule, M. Nunkesser, and A. Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State-of-The-Art Cardinality Estimation Algorithm. *Proc. of EDBT* (2013).
[27] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. *Proc. of ACM SIGCOMM* (2017).
[28] Q. Huang, P. P. C. Lee, and Y. Bao. 2018. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. *Proc. of ACM SIGCOMM* (August 2018), 576 – 590.
[29] A. Kumar, M. Sung, J. Xu, and J. Wang. 2004. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. *Proc. of ACM SIGMETRICS* (June 2004).
[30] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. 2004, A journal version was published in IEEE JSAC, 24(12):2327-2339, December 2006. Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement. *Proc. of IEEE INFOCOM* (March 2004, A journal version was published in IEEE JSAC, 24(12):2327-2339, December 2006).
[31] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. 2006. Data Streaming Algorithms for Estimating Entropy of Network Traffic. *Proc. of SIGMETRICS/Performance* (2006).
[32] T. Li, S. Chen, and Y. Ling. 2011. Fast and Compact Per-Flow Traffic Measurement through Randomized Counter Sharing. *IEEE INFOCOM* (2011).
[33] Y. Li, R. Miao, C. Kim, and M. Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. *in Proc. of USENIX NSDI* (2016).
[34] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. *Proc. of ACM Sigcomm* (2016).
[35] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. 2008. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. *Proc. of ACM SIGMETRICS* (June 2008).
[36] Y. Lu and B. Prabhakar. 2009. Robust Counting Via Counter Braids: An Error-Resilient Network Measurement Architecture. *Proc. of IEEE INFOCOM* (April 2009).
[37] P. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. 2002. Controlling High Bandwidth Aggregates in the Network. *Computer Communications Review* 32, 3 (July 2002), 62–73.
[38] J. Mai, C. N. Chuan, A. Sridharan, T. Ye, and H. Zang. 2006. Is sampled data sufficient for anomaly detection? *Proc. of ACM IMC* (2006).

[39] G. Manku and R. Motwani. 2002. Approximate Frequency Counts over Data Streams. *Proc. of VLDB* (August 2002).
[40] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. 2003. Internet Quarantine: Requirements for Containing Self-Propagating Code. *Proc. of IEEE INFOCOM* (April 2003).
[41] D. Moore, G. Voelker, and S. Savage. 2001. Inferring Internet Denial of Service Activity. *Proc. of USENIX Security Symposium'2001* (August 2001).
[42] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. 2014. DREAM: Dynamic Resource Allocation for Software-defined Measurement. *in Proc. of ACM SIGCOMM* (2014).
[43] UF Networklab. Online. The Source Codes of Generalized Sketch Families. https://github.com/mcynever/GeneralizedSketchFamilies
[44] Nvidia. Online. Nvidia cuda c programming guide, version 10.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[45] OVS. Online. Open vSwitch. https://www.openvswitch.org/
[46] K. Park and H. Lee. 2001. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. *Proc. of ACM SIGCOMM'2001* (August 2001).
[47] D. Plonka. 2000. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. *Proc. of USENIX LISA* (2000).
[48] M. Roesch. 1999. Snort — Lightweight Intrusion Detection for Networks. *Proc. of 13th Systems Administration Conference, USENIX* (1999).
[49] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. 2004. Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams. *Proc. of IMC* (2004).
[50] S. Staniford, V. Paxson, and N. Weaver. 2002. How to 0wn the Internet in Your Spare Time. *Proc. of USENIX Security Symposium* (August 2002).
[51] UCSD. 2015. CAIDA UCSD Anonymized 2015 Internet Traces on Jan. 17. http://www.caida.org/data/passive/passive_2015_dataset.xml.
[52] S. Venkatataman, D. Song, P. Gibbons, and A. Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. *Proc. of NDSS* (February 2005).
[53] H. Wang, D. Zhang, and K. G. Shin. 2002. SYN-dog: Sniffing SYN Flooding Sources. *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS'02)* (July 2002).
[54] K. Whang, B. T. Vander-Zanden, and H. M. Taylor. 1990. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems* 15, 2 (1990), 208–229.
[55] Wikipedia. Online. Chebyshev inequality. https://en.wikipedia.org/wiki/Chebyshev%27s_inequality
[56] Q. Xiao, S. Chen, M. Chen, and Y. Ying. 2015. Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing. *in Proc. of ACM SIGMETRICS* (2015).
[57] Q. Xiao, S. Chen, Y. Zhou, M. Chen, J. Luo, T. Li, and Y. Ling. 2017. Cardinality Estimation for Elephant Flows: A Compact Solution based on Virtual Register Sharing. *IEEE/ACM Transactions on Networking* (2017).
[58] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. *Proc. of ACM SIGCOMM* (August 2018).
[59] M. Yoon, T. Li, S. Chen, and J. Peir. 2009. Fit a Spread Estimator in Small Memory. *Proc. of IEEE INFOCOM* (April 2009).
[60] M. Yoon, T. Li, S. Chen, and J. Peir. 2011. Fit a Compact Spread Estimator in Small High-Speed Memory. *IEEE/ACM Transactions on Networking* 19, 5 (October 2011), 1253–1264.
[61] M. Yu, L. Jose, and R. Miao. 2013. Software Defined Traffic Measurement with OpenSketch. *Proc. of USENIX Symposium on Networked Systems Design and Implementation* (2013).
[62] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. 2004. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Application. *Proc. of ACM SIGCOMM IMC* (October 2004).
[63] Y. Zhou, Y. Zhou, M. Chen, and S. Chen. 2017. Persistent Spread Measurement for Big Network Data Based on Register Intersection. *Proc. of ACM SIGMETRICS* (2017).

APPENDIX I. PSEUDO CODE

Algorithm 8 *vQuery_Plugin*(L_f)

Input: estimator L_f
Action: size/spread estimate $\hat{k}$

switch (type of estimator L_f)
case counter**:**
  $\hat{k} := 0$
  for $j = 0..m-1$ **do**
    $\hat{k} := \hat{k} + L_f[j]$
  end for
case bitmap**:**
  $Z = 0$
  for $j = 0..b-1$ **do**
    if $L_f[j] = 0$ **then** $Z := Z + 1$
  end for
  $\hat{k} := -b \ln(\frac{Z}{b})$
case FM**:**
  $p := 0$
  for $j = 0..m-1$ **do**
    $p :=$ $p+$ no. of consecutive leading ones in $L_f[j]$
  end for
  $p = \frac{p}{m}$; $\hat{k} = m2^p/\varphi$
case HLL**:**
  $\alpha_m := 0.7213/(1 + 1.079/m)$; $sum = 0$
  for $j = 0..m-1$ **do**
    $sum := sum + 2^{-L_f[j]}$
  end for
  $\hat{k} := \alpha_m m^2/sum$
end switch
return $\hat{k}$

APPENDIX II. PROOF OF THEOREM 1

Proof. From (7), $x = k + n$, where x is the number of elements recorded in the logical estimator L_f, consisting of k elements from flow f and n noise elements from other flows, with k being a constant and n being a random variable. For each instance value of n, from (10), we have

$$(k+n)(1-\delta) \le E(\hat{x}) \le (k+n)(1+\delta). \tag{12}$$

Over the distribution of n, we have

$$(k+E(n))(1-\delta) \le E(\hat{x}) \le (k+E(n))(1+\delta). \tag{13}$$

Applying (8) to (13), we have

$$(k+\frac{X-k}{w})(1-\delta) \le E(\hat{x}) \le (k+\frac{X-k}{w})(1+\delta). \tag{14}$$

Consider the super estimator L_F that records all X elements. From (10), we have

$$X(1-\delta) \le E(\hat{X}) \le X(1+\delta). \tag{15}$$

Because $\hat{k} = \hat{x} - \hat{n}$ and $\hat{n} = \frac{\hat{X}}{w}$,

$$E(\hat{k}) = E(\hat{x}) - \frac{E(\hat{X})}{w}. \tag{16}$$

From (13), (15) and (16), we have

$$k(1 - \frac{1}{w} - \delta(1 - \frac{1}{w} + \frac{2X}{wk})) \leq E(\hat{k}) \leq k(1 - \frac{1}{w} + \delta(1 - \frac{1}{w} + \frac{2X}{wk})). \tag{17}$$

Let $\epsilon = \delta(1 - \frac{1}{w} + \frac{2X}{wk})$. Eq. (17) can be rewritten as

$$k(1 - \epsilon - \frac{1}{w}) \leq E(\hat{k}) \leq k(1 + \epsilon - \frac{1}{w}). \tag{18}$$

□

APPENDIX III. PROOF OF THEOREM 2

Proof. Below we derive $Var(\hat{k})$ for vSkt(counter), vSkt(bitmap), vSkt(FM) and vSkt(HLL), respectively.

1. Case of vSkt(counter)

Consider the logical estimator L_f of an arbitrary flow f. Recall that $x = k + n$, where x is the estimate from L_f (see Line 3-6 of Alg. 8), k is the actual size of the flow, and n is the noise, which follows a Binomial distribution, $n \sim Bino(X-k, \frac{1}{w})$, because L_f consists of m counters randomly selected from the wm counters in U. Hence, $Var(n) = \frac{X-k}{w}(1 - \frac{1}{w}) \approx \frac{X}{w}(1 - \frac{1}{w})$ when $k \ll X$, where X is the sum of the sizes of all flows. For vSkt(Counter), $\hat{X} = X$, which is a constant, $\hat{x} = x$, and thus $Var(\hat{x}) = Var(x) = Var(n) \approx \frac{X}{w}(1 - \frac{1}{w})$. We have

$$Var(\hat{k}) = Var(\hat{x} - \frac{\hat{X}}{w}) = Var(\hat{x}) = \frac{X}{w}(1 - \frac{1}{w}) \tag{19}$$

2. Case of vSkt(bitmap)

Because $\hat{k} = \hat{x} - \hat{n}$, we have

$$Var(\hat{k}) = Var(\hat{x}) + Var(\hat{n}) + 2(E(\hat{x})E(\hat{n}) - E(\hat{x}\hat{n})). \tag{20}$$

Let V_f be the percentage of bits in the logical estimator L_f that are zeros. Before we derive $Var(\hat{k})$, we need to derive $Var(V_f)$ first. Consider an arbitrary bit $L_f[i], 0 \leq i < m$, in the logical estimator L_f where elements from f and other flows (noise) randomly distribute among its m bits. For the k elements from flow f, the probability that $L_f[i]$ is not chosen to record any of them is $(1 - \frac{1}{m})^k$. The $(X - k)$ noise elements from other flows are randomly distributed among all wm physical bits. Hence, those recorded by the m bits in L_f follow a Binomial distribution, i.e., $n \sim Bino(X - k, \frac{1}{w})$. Therefore, the probability that $L_f[i]$ does not record any noise element is

$$\begin{aligned}
&\sum_{0}^{X-k} \binom{X-k}{i} (\frac{1}{w})^i (1 - \frac{1}{w})^{X-k-i} (1 - \frac{1}{m})^i \\
&= \sum_{0}^{X-k} \binom{X-k}{i} (1 - \frac{1}{w})^{X-k-i} (\frac{1}{w} - \frac{1}{mw})^i \\
&= (1 - \frac{1}{w} + \frac{1}{w} - \frac{1}{mw})^{X-k} = (1 - \frac{1}{mw})^{X-k}.
\end{aligned} \tag{21}$$

Let $\mathcal{A}_i$ be the event that $L_f[i]$ does not record any element and thus remains 0. Let $1_{\mathcal{A}_i}$ be the corresponding indictor variable, which is 1 when $\mathcal{A}_i$ happens and 0 otherwise. We have

$$Prob(\mathcal{A}_i) = (1 - \frac{1}{mw})^{X-k}(1 - \frac{1}{m})^k. \tag{22}$$

The expected number of zero bits in L_f, denoted as Z_f, is

$$\begin{aligned} E(Z_f) &= \sum_{j=0}^{m-1} 1_{\mathcal{A}_i} = \sum_{j=0}^{m-1} Prob(\mathcal{A}_i) \\ &= \sum_{i=0}^{m-1} (1 - \frac{1}{mw})^{X-k}(1 - \frac{1}{m})^k \\ &\approx me^{-(\frac{X}{mw} + \frac{k}{m})}. \end{aligned} \tag{23}$$

Because $V_f = \frac{Z_f}{m}$, we have

$$E(V_f) = e^{-(\frac{X}{mw} + \frac{k}{m})}. \tag{24}$$

Consider any two bits, $L_f[i]$ and $L_f[j]$, $i \neq j$. The probability for both of them to be zeros is

$$Prob(\mathcal{A}_i \cap \mathcal{A}_j) = (1 - \frac{2}{mw})^{X-k}(1 - \frac{2}{m})^k. \tag{25}$$

Therefore, we have

$$\begin{aligned} E((V_f)^2) &= \frac{1}{m^2} E((\sum_{i=0}^{m-1} 1_{\mathcal{A}_i})^2) \\ &= \frac{1}{m^2} E(\sum_{i=0}^{m-1} (1_{\mathcal{A}_i})^2) + \frac{2}{m^2} E(\sum_{i=0}^{m} \sum_{j=0}^{i-1} (1_{\mathcal{A}_i} 1_{\mathcal{A}_j})) \\ &= \frac{1}{m}(1 - \frac{1}{mw})^{X-k}(1 - \frac{1}{m})^k \\ &\quad + \frac{m-1}{m}(1 - \frac{2}{mw})^{X-k}(1 - \frac{2}{m})^k. \end{aligned} \tag{26}$$

Applying (24) and (26) to $Var(V_f) = E(V_f - E(V_f))^2$, we have

$$\begin{aligned}
Var(V_f) &= E(V_f - E(V_f))^2 = E((V_f)^2) - (E(V_f))^2 \\
&= \frac{1}{m}(1 - \frac{1}{mw})^{X-k}(1 - \frac{1}{m})^k \\
&\quad + \frac{m-1}{m}(1 - \frac{2}{mw})^{X-k}(1 - \frac{2}{m})^k \\
&\quad - (1 - \frac{1}{mw})^{2(X-k)}(1 - \frac{1}{m})^{2k} \\
&= \frac{m-1}{m}((1 - \frac{2}{mw})^{X-k}(1 - \frac{2}{m})^k \\
&\qquad - (1 - \frac{1}{mw})^{2(X-k)}(1 - \frac{1}{m})^{2k}) \\
&\quad + (1 - \frac{2}{m})^{X-k}(1 - \frac{2}{m})^k \\
&\quad - (1 - \frac{1}{mw})^{2(X-k)}(1 - \frac{1}{m})^{2k} \\
&\approx \frac{1}{m}(e^{-(\frac{X}{mw} + \frac{k}{m})} - e^{-2(\frac{X}{mw} + \frac{k}{m})}) \\
&\quad + e^{-2(\frac{X-k}{m} + \frac{k}{m})}(\frac{-k}{m^2}) \\
&\approx \frac{1}{m}(e^{-(\frac{X}{mw} + \frac{k}{m})} - e^{-2(\frac{X}{mw} + \frac{k}{m})} \\
&\qquad - \frac{k}{m}e^{-2(\frac{X}{mw} + \frac{k}{m})}).
\end{aligned} \tag{27}$$

From [54], the estimate of x from the logical estimator L_f is given as

$$\hat{x} = -m \ln(V_f). \tag{28}$$

Hence,

$$E(\hat{x}) = E(-m \ln(V_f)). \tag{29}$$

By using the Taylor Series of $\ln(V_f)$ and applying (24) and (27) to (29), we have

$$\begin{aligned}
E(\hat{x}) &= E(-m \ln(V_f)) \\
&\approx E(\ln(E(V_f) + \frac{V_f - E(V_f)}{E(V_f)} - \frac{(V_f - E(V_f))^2}{2E(V_f)^2}) \\
&\approx m(\frac{X}{mw} + \frac{k}{m} + \frac{1}{2E(V_f)^2}E((V_f - E(V_f))^2)) \\
&= m(\frac{X}{mw} + \frac{k}{m} + \frac{e^{\frac{X}{mw} + \frac{k}{m}} - 1 - \frac{k}{m}}{2m})
\end{aligned} \tag{30}$$

where

$$\ln(V_f) = \ln(E(V_f)) + \frac{V_f - E(V_f)}{E(V_f)} - \frac{(V_f - E(V_f))^2}{2E(V_f)^2} + ... \tag{31}$$

Similarly, using the Taylor Series of $\ln(V_f)$ and keeping the first two terms, we derive the variance of $\hat{x}$ as

$$\begin{aligned} Var(\hat{x}) &= Var(-m\ln(V_f)) \\ &\approx m^2 Var(\ln(E(V_f) + \frac{V_f - E(V_f)}{E(V_f)}) \\ &\approx m^2 Var(\frac{X}{mw} + \frac{k}{m} - \frac{V_f - E(V_f)}{E(V_f)}) \\ &= \frac{m^2}{E(V_f)^2} Var(V_f) \\ &\approx m(e^{(\frac{X}{mw} + \frac{k}{m})} - \frac{k}{m} - 1). \end{aligned} \tag{32}$$

Next we derive the mean and variance of $\hat{n}$, which is equal to $\frac{\hat{X}}{w}$. From [54],

$$\hat{X} = -mw\ln(V_F). \tag{33}$$

Hence, $\hat{n} = -m\ln(V_F)$. We have

$$E(\hat{n}) = E(-m\ln(V_F)). \tag{34}$$

By using the following Taylor Series

$$\ln(V_F) = \ln(E(V_F)) + \frac{V_F - E(V_F)}{E(V_F)} - \frac{(V_F - E(V_F))^2}{2E(V_F)^2} + ..., \tag{35}$$

we have

$$\begin{aligned} E(\hat{n}) &= E(-m\ln(V_F)) \\ &\approx m(\frac{X}{mw} + \frac{1}{2E(V_F)^2} E((V_F - E(V_F))^2)) \\ &= m(\frac{X}{mw} + \frac{1}{2E(V_F)^2} Var(V_F)). \end{aligned} \tag{36}$$

Let V_F be the percentage of bits in the super estimator L_F that are zeros. From [54], its mean and variance are

$$\begin{aligned} E(V_F) &\approx e^{-\frac{X}{mw}}, \\ Var(V_F) &\approx \frac{e^{-\frac{X}{mw}}}{mw}(1 - (1 + \frac{X}{mw})e^{-\frac{X}{mw}}). \end{aligned} \tag{37}$$

Applying (37) to (36), we have

$$\begin{aligned} E(\hat{n}) &= E(-m\ln(V_F)) \\ &= m(\frac{X}{mw} + \frac{e^{\frac{X}{mw}} - 1 - \frac{X}{mw}}{2mw}). \end{aligned} \tag{38}$$

Using the Taylor series (35) and keeping the first two terms, we have

$$\begin{aligned} Var(\hat{n}) &= Var(-m\ln(V_F)) \\ &\approx m^2 Var(\frac{X}{mw} - \frac{V_F - E(V_F)}{E(V_F)}) \\ &= \frac{m^2}{E(V_F)^2} Var(V_F) \\ &\approx \frac{m}{w}(e^{\frac{X}{mw}} - 1 - \frac{X}{mw}). \end{aligned} \tag{39}$$

Furthermore, by using the Taylor series (31) and (35), we have

$$\begin{aligned} E(\hat{x}\hat{n}) &= E(m^2 \ln(V_f)\ln(V_F)) \\ &\approx m^2 E((\ln(E(V_f)) + \frac{V_f - E(V_f)}{E(V_f)}) \cdot \\ &(\ln(E(V_F)) + \frac{V_F - E(V_F)}{E(V_F)})) \end{aligned} \tag{40}$$

By applying (29) and (37) to (40), we have

$$\begin{aligned} E(\hat{x}\hat{n}) \approx m^2 (& \frac{X}{mw}(\frac{X}{mw} + \frac{k}{m} + \frac{e^{\frac{X}{mw}+\frac{k}{m}} - 1 - \frac{k}{m}}{2m}) \\ & + \frac{\frac{X}{mw} + \frac{k}{m}}{mw}(X + \frac{e^{\frac{X}{mw}} - \frac{X}{mw} - 1}{2}) \\ & - \frac{X}{mw}(\frac{X}{mw} + \frac{k}{m}) \\ = m^2 (& \frac{X}{mw}(\frac{X}{mw} + \frac{k}{m}) + \frac{X(e^{\frac{X}{mw}+\frac{k}{m}} - 1 - \frac{k}{m})}{2m^2 w} \\ & + \frac{(\frac{X}{mw} + \frac{k}{m})(e^{\frac{X}{mw}} - \frac{X}{mw} - 1)}{2mw})) \end{aligned} \tag{41}$$

Finally, by applying (30), (32), (38), (39) and (41) to (20), we have

$$\begin{aligned} Var(\hat{k}) \approx & \, m(e^{\frac{X}{mw}+\frac{k}{m}} - 1 - \frac{k}{m}) + \frac{m(e^{\frac{X}{mw}} - \frac{X}{mw} - 1)}{w} \\ & + \frac{(e^{\frac{X}{mw}+\frac{k}{m}} - 1 - \frac{k}{m})(e^{\frac{X}{mw}} - \frac{X}{mw} - 1)}{2w}. \end{aligned} \tag{42}$$

3. Case of vSkt(HLL)

Because $\hat{k} = \hat{x} - \hat{n}$ and $\hat{n} = \frac{\hat{X}}{w}$, we have

$$\begin{aligned} Var(\hat{k}) &= Var(\hat{x} - \hat{n}) = Var(\hat{x}) - 2Cov(\hat{x}, \hat{n}) + Var(\hat{n}) \\ &= Var(\hat{x}) - 2Cov(\hat{x}, \frac{\hat{X}}{w}) + \frac{Var(\hat{X})}{w^2}, \\ &= Var(\hat{x}) - \frac{2}{w}Cov(\hat{x}, \hat{X}) + \frac{Var(\hat{X})}{w^2}, \end{aligned} \tag{43}$$

where $\hat{x}$ is the estimate from the logical estimator L_f and $\hat{X}$ is the estimate from the super estimator L_F. Assuming that the contribution of elements recorded in a single logical

estimator L_f is negligible to the super estimator L_F, which is true when $k \ll X$ and w is large, we have $Cov(\hat{x}, \hat{X}) = 0$. Therefore,

$$\begin{aligned} Var(\hat{k}) &= Var(\hat{x}) + \frac{Var(\hat{X})}{w^2}, \\ &= E(\hat{x}^2) - (E(\hat{x}))^2 + \frac{Var(\hat{X})}{w^2}. \end{aligned} \tag{44}$$

According to [21], for a logical estimator L_f which records x elements, the expectation and variation of its estimate $\hat{x}$ are given as

$$E(\hat{x}) = x(1 + \epsilon_1(x) + o(1)), \tag{45}$$

$$\sqrt{Var(\hat{x})} = x(\frac{\beta_m}{\sqrt{m}} + \epsilon_2(x) + o(1)), \tag{46}$$

where $\epsilon_1, \epsilon_2 \leq 0.00005$ when $m \geq 16$ and $\beta_m = 1.046$ when $m \geq 128$.

Consider the logical estimator L_f of m registers which records elements from both flow f and other flows (noise). Since all $(X - k)$ noise elements are randomly distributed among $m \times w$ registers, the distribution of the noise elements recorded by L_f follows a Binomial distribution, i.e., $n \sim Bino(X - k, \frac{1}{w})$. Because $n = x - k$, we have

$$Prob\{x - k = i\} = \binom{X-k}{i}(\frac{1}{w})^i(1 - \frac{1}{w})^{X-k-i}. \tag{47}$$

From (45),

$$E(\hat{x}|x - k = i) = (k + i)(1 + \epsilon_1(k + i) + o(1)) \approx k + i \tag{48}$$

The expectation of $\hat{x}$ can be derived as

$$\begin{aligned} E(\hat{x}) &= \sum_{i=0}^{X-k} E(\hat{x}|x - k = i)Prob(x - k = i) \\ &\approx \sum_{i=0}^{X-k} (k + i)\binom{X-k}{i}(\frac{1}{w})^i(1 - \frac{1}{w})^{X-k-i} \\ &= k + \frac{X}{w}. \end{aligned} \tag{49}$$

From (46),

$$\begin{aligned} \sqrt{Var(\hat{x}|x - k = i)} &= (k + i)(\frac{\beta_m}{\sqrt{m}} + \epsilon_2(k + i) + o(1)) \\ &= (k + i)\frac{\beta_m}{\sqrt{m}} \approx (k + i)\frac{1.04}{\sqrt{m}}, \end{aligned} \tag{50}$$

where $\beta_m = 1.04$ and $m >= 128$. So we have

$$Var(\hat{x}|x - k = i) \approx \frac{1.04^2}{m}(k + i)^2. \tag{51}$$

Therefore,

$$\begin{aligned} E(\hat{x}^2|x = k + i) &= Var(\hat{x}|x - k = i) + (E(\hat{x}|x - k = i))^2 \\ &\approx ((k + i)\frac{1.04}{\sqrt{m}})^2 + (k + i)^2 \\ &= (\frac{1.04^2}{m} + 1)(k + i)^2. \end{aligned} \tag{52}$$

$$
\begin{aligned}
E(\hat{x}^2) &= \sum_{i=0}^{X} E(\hat{x}^2|x = k+i)Prob(x-k=i) \\
&\approx \sum_{i=0}^{X} (\frac{1.04^2}{m}+1)(k+i)^2 \binom{X}{i} (\frac{1}{w})^i (1-\frac{1}{w})^{X-i} \\
&= (\frac{1.04^2}{m}+1)(k^2+2kE(x-k)+E((x-k)^2)) \\
&= (\frac{1.04^2}{m}+1)((k+\frac{X}{w})^2+\frac{X}{w}(1-\frac{1}{w})).
\end{aligned} \tag{53}
$$

For the super flow F which is recorded in the super estimator L_F of m HLL registers, from [21],

$$
Var(\hat{X}) \approx \frac{1.04^2}{m} X^2. \tag{54}
$$

Finally, by applying (49), (53) and (54) to (44), we have

$$
\begin{aligned}
Var(\hat{k}) \approx & \frac{1.04^2}{m}(k+\frac{X}{w})^2 + \frac{1.04^2}{mw^2}X^2 \\
& + (\frac{1.04^2}{m}+1)\frac{X}{m}(1-\frac{1}{w}).
\end{aligned} \tag{55}
$$

4. Case of vSkt(FM)

For vSkt(FM), following a process similar to vSkt(HLL) and applying the analysis results from [22], we can derive the variance of its estimate $\hat{k}$ as

$$
\begin{aligned}
Var(\hat{k}) \approx & \frac{0.78^2}{m}(k+\frac{X}{w})^2 + \frac{0.78^2}{mw^2}X^2 \\
& + (\frac{0.78^2}{m}+1)\frac{X}{m}(1-\frac{1}{w}).
\end{aligned} \tag{56}
$$

□

Received August 2019; revised October 2019; accepted October 2019