Jellyfish: Locality-sensitive Subflow Sketching

Yongquan Fu, Lun An, Siqi Shen, Kai Chen, Pere Barlet-Ros
National University of Defense Technology, Changsha, China
School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China
Cyberspace Security Research Center, Peng Cheng Laboratory, China
Xiamen University
SING Lab, HKUST
Barcelona Neural Networking Center, Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*—To cope with increasing network rates and massive traffic volumes, sketch-based methods have been extensively studied to trade accuracy for memory scalability and storage cost. However, sketches are sensitive to hash collisions due to skewed keys in real world environment, and need complicated performance control for line-rate packet streams.**

We present Jellyfish, a locality-sensitive sketching framework to address these issues. Jellyfish goes beyond network flow-based sketching towards fragments of network flows called subflows. First, Jellyfish splits consecutive packets from each network flow to subflow records, which not only reduces the rate contention but also provides intermediate subflow representations in form of truncated counters. Next, Jellyfish maps similar subflow records to the same bucket array and merges those from the same network flow to reconstruct the network-flow level counters. Real-world trace-driven experiments show that Jellyfish reduces the average estimation errors by up to six orders of magnitude for per-flow queries, by six orders of magnitude for entropy queries, and up to ten times for heavy-hitter queries.

***Index Terms*—subflow, sketch, hash collision, clustering, heavy hitter**

I. Introduction

Network measurement is of paramount importance for traffic engineering, network diagnosis, network forensics, intrusion detection and prevention, and network routing in clouds and data centers, which need a variety of measurement tasks, such as delay, flow size estimation, flow distribution, and heavy hitter detection. Network-flow monitoring is challenging due to the ever-increasing line rates, massive traffic volumes, and large number of active flows.

Traffic analysis tasks require advanced data structures and algorithms. Many space- and time-efficient approaches have been proposed in the literature, e.g., traffic sampling (e.g., Netflow, sflow), traffic counting [1], [2], [3], [4], traffic sketching [5], [6], [7], [8], [9], [10], [11].

Traffic sketching is an increasingly popular method compared to traffic sampling and traffic counting, due to their better trade-off between space requirements and capability of ingesting all packet records to a constant-size data structure. The sketch maps each incoming packet to a bucket indexed by the hash of the corresponding network flow identifier. Thus all packets from the same network flow would be mapped to the same bucket, which approximately preserves the network-flow counter.

Due to the hashing collisions caused by mapping multiple keys to the same sketch entries, the sketch-based network monitoring process incurs a high degree of approximation errors. Hash collisions are due to the randomness of the hash functions, thus it is impossible to avoid them. Moreover, the probability of hash collisions increases as more keys are hashed into the fixed-size data structure.

To reduce approximation errors, the common approach is to either increment the size of the sketch structure, or to maintain multiple independent sketching instances for the purpose of choosing the sketch entries with the fewest hash collisions. Yet, real-world traffic distributions are usually non-uniform, so that the approximation errors are likely to be amplified by the long tails of network-flow distributions.

Most severe approximation errors are mainly caused by the traffic-distribution-oblivious hashing process, thus a natural question is whether we can reduce the approximation errors by replacing key-based hashing process with locality-sensitive methods. Several approaches [12], [13] proposed to separate large ones from the rest to reduce the peak estimation error for the rest. LSS [14] proposed to map similar network counters to the same bucket array in order to reduce the approximation error. Unfortunately, LSS needs to keep all active network flows in the memory, in order to update the mapping of network flows to the nearest clusters in real time as the network flow counters keep increasing, which is challenging with respect to large-scale network traffic.

This paper exploits the *subflow*, an aggregated record for a subset of packets from the same network flow, to build Jellyfish, a locality-sensitive sketching framework. Jellyfish directly maps similar subflow counters to the same bucket array and estimates each network-flow counter by the sum of estimated subflow counters from the same network flow.

The granularity of the subflow can be adjusted with a flexible threshold parameter that bounds the maximal subflow counter. Reducing the thresholds smooths the subflow distributions, but increases the number of subflow records for reconstructing the network-flow level counters. As a result, we can set suitable subflow thresholds for different network-flow distributions.

Replacing network-flow level sketching with subflow-level sketching is non-trivial. A strawman approach for subflow-based sketching is to use conventional sketching methods

TABLE I
KEY NOTATIONS.

Notation	Meaning
N	Number of keys
k	Number of cluster centers
m	Number of buckets
τ	Subflow threshold
f_s	Cardinality of subflows of a network flow

such as count-min [15] or count-sketch [16] methods. Our experiments in Subsection III-C show that these sketching methods still incur a high degree of approximation errors, since truncated subflow counters are still non-uniform. As a result, subflow sketching requires locality-sensitive methods.

To be locality-sensitive for online subflow counters, Jellyfish directly clusters similar subflow records to the same bucket array, and postpones the reconstruction of the network flow counter during the query process. Jellyfish generates subflow records from packet streams with a high-performance hash table, then clusters similar subflow counters to the same bucket array and reconstructs the network-flow counter by merging the queried results of subflow records from this network flow.

We perform extensive evaluation in Section V. Real-world trace-driven experiments show that Jellyfish outperforms state-of-the-art sketching methods, with up to 10^6X reduction in average relative errors for per-flow queries, 10^6X reduction in average relative errors for entropy queries, and up to 10X reduction in average F1 scores for heavy-hitter queries.

We summarize our contributions as follows: (i) We present a locality-sensitive subflow sketching method Jellyfish. (ii) We present a subflow based network monitoring framework that generates tunable subflow records from packet streams and decouples the sketch structure from the ingestion components to adapt it to the line rates. (iii) We conduct extensive experiments with real-world data sets to show that the proposed sketch method obtains accurate and robust estimation results.

The rest of the paper is organized as follows. Sec. II summarizes related studies on sketch-based network monitoring methods and provides the problem model on the sketch-based network flow monitoring process. Sec. III next presents a practical locality-sensitive sketch. Sec. IV presents the implementation details. Sec. V conducts extensive performance evaluation with real-world datasets. We finally conclude in Sec. VI. Table I lists key notations.

II. BACKGROUND AND RELATED WORK

We present the background for the network monitoring process, and the related work that are most related with us. We next analyze the challenges of the sketching approaches.

A. Background

A network flow is typically represented as a key-value pair, where the key is defined as the composition of several essential packet-header fields, and the value is defined as the flow's current statistics, e.g., number of packets or flow bytes.

For each incoming packet, a sketch-based monitor inspects the packet header to extract the key and calculate the packet's value, then inserts this record to the sketch data structure. A sketch-based monitoring application typically comprises an *ingestion* component that intercepts incoming packets from the physical network interface and generates key-value input for the sketch, a *sketching* component that feeds the key-value input to a sketch structure that approximates these key-value pairs with one or multiple hash based bucket arrays.

There are two schemes for network monitoring on network flows. For the sequence based sliding window, each interval keeps at most N flow records, and a new interval is generated afterwards; while for a time based window, each interval records the packets during a fixed time period, and a new interval is created after the interval ends.

B. Related Work

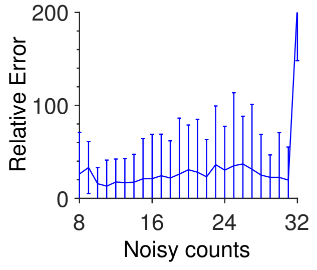
To reduce the approximation error, a popular trick is to choose the least affected bucket from multiple copies of independent bucket arrays as the estimator. UnivMon [9] uses an array of sketch counters to meet generic monitoring tasks. SketchVisor [17] augments the sketch with a fast-path ingestion path to tolerate bursty traffic. ElasticSketch [12] keeps heavy hitters separately with a hash table, and puts the rest of items to a count-min sketch. Thus it is less sensitive to heavy hitters compared to prior sketch structures [16], [15], [18]. SketchLearn [13] separates large flows from the rest based on inferred flow distributions, which incurs additional processing delay for each packet. Nitrosketch [19] reduces the insertion frequency to relieve the processing delay, but introduces uncertainty on the sketching results. OmniMon [20] seeks full accuracy and resource efficiency over collaborated network entities. LSS[14] applies a cluster-preserving approach to reduce the estimation error, by guaranteeing that each network flow is always mapped to the closest cluster. Thus, to process packet streams like traditional synopsis, LSS maintains an in-memory cache for active network flows and dynamically adjusts the clusters for these network flows. Our work dramatically improves the prediction accuracy and memory efficiency using a subflow clustering based sketch method, which avoids the need of adjusting the buckets for active network flows.

C. Sketch Challenges

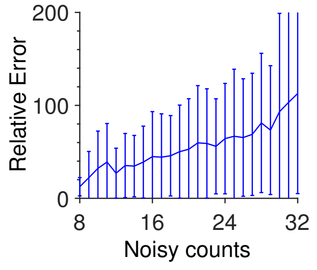
A sketch-based streaming should meet monitoring accuracy and performance needs for network-wide streams in cloud data center networks.

(i) **Approximation error**: Sketch-based methods incur a degree of approximation errors.

We give an empirical test of the estimation error of count-min method [15] and count-sketch method [16] with real-world trace data set, introduced in subsection V-A. Figure 1 plots the relative error of queried items with respect to the number of items hashed into the corresponding bucket. We see that even a relatively small number of hash collisions significantly degrade the prediction accuracy. As a result, we



(a) Count-sketch method [16]



(b) Count-sketch method [15]

Fig. 1. The average and the standard deviation of the relative errors of queried items with respect to the number of items inserted into the corresponding bucket on the CAIDA data set. We use three bucket arrays for a sketch, and set the ratio between the number of buckets and the number of network flows to 0.05.

need an elegant approach to reduce the variance of the keys within each bucket array, while preserving the simplicity of sketching operations.

Generally, we can quantify the expected number of noisy buckets that have hash collisions. Assume that each key is mapped to a bucket in each bank uniformly at random with the hash function. As Lemma 1 shows that, the probability of hash collisions increases fast with decreasing ratios between the number of buckets and the number of inserted keys.

Lemma 1. *Let m denote the number of buckets, N the number of unique keys. For a sketch with c banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$, the expected percent of noisy buckets is $1 - e^{-cN/m} - \frac{cN}{m} \cdot e^{-c(N-1)/m}$.*

Proof. For a sketch with one bucket array that consists of m buckets, the expected number of keys per bucket amounts to $\frac{N}{m}$. The expected number of empty buckets is: $\sum_i \left(1 - \frac{1}{m}\right)^N = m\left(1 - \frac{1}{m}\right)^N \approx me^{-N/m}$. Similarly, the expected number of buckets with one key amounts to: $\sum_i \binom{N}{1} \left(\frac{1}{m}\right)\left(1 - \frac{1}{m}\right)^{N-1} \approx Ne^{-(N-1)/m}$. As a result, the expected percent of buckets that contain at least two keys is $\left(m - me^{-N/m} - Ne^{-(N-1)/m}\right) \cdot \frac{1}{m}$. The expected percent of noisy buckets is $1 - e^{-N/m} - \frac{N}{m} \cdot e^{-(N-1)/m}$.

For a sketch with c banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$, each bucket array still receives N keys. Thus following the same derivation, we have that the corresponding expected percent of noisy buckets is $1 - e^{-cN/m} - \frac{cN}{m} \cdot e^{-c(N-1)/m}$. □

(ii) **Sketching efficiency**: Sketch-based methods are typically coupled with the packet rate, not the flow rate. Although a sketch only produces flow-level estimation results, existing monitoring applications feed packet-granularity streams to the sketch. The need of coping with line-rate packets increases the resource contentions of the sketch structure with colocated deployed systems [17]. As the network is getting faster, more packets must be inspected for the same amount of time, which implies that the sketch's space and time complexity must be tightly controlled. The packet rate is typically much smaller than the network-flow rate. For example, CAIDA reported that on 11-15, 2018, the mean transmission rates of two directions for the equinix path [21] reach at 620.41 $\times 10^3$, and 1.56 $\times 10^6$ packets per second, respectively; while the number of flows per second were just 31.80 $\times 10^3$ and 91.02 $\times 10^3$, respectively.

This paper deals with the sketching inefficiency and the approximation errors with a subflow sketching framework, which goes beyond network flow-based sketching towards fragments of network flows.

III. Jellyfish: Subflow Sketching

Having presented the challenges of existing sketching methods, we next present a locality-sensitive subflow sketch called Jellyfish.

A. Overview

Figure 2 shows the overall framework of the Jellyfish based network monitoring process. Jellyfish generates subflow records by aggregating the counters of a subset of consecutive packets from the same network flow. Next, Jellyfish clusters similar subflow records to bucket arrays with respect to an online subflow-clustering model. Specifically, Jellyfish puts each subflow record into a bucket array indexed by the closest cluster center towards this record. Next, Jellyfish selects a random bucket from this bucket array, by hashing the subflow key with one hash function, and accumulates the subflow counter to this bucket. Finally, Jellyfish reconstructs the counters for each network flow by estimating each subflow record belonging to this network flow and merging those to the final estimation result.

From Figure 2, we see that the approximation results closely match the original network flow counters. Moreover, Jellyfish does not need an in-memory cache of historical samples, or a real-time update policy to dynamically adjust the clustering positions of evolving network flow counters. We present implementation details of the subflow clustering model in Sec. IV.

Figure 3 summarizes key functions in the subflow-based network monitoring process with ingestion, membership, insertion and query functions. The ingestion function produces subflow records. The membership function keeps the mapping of subflow records to bucket arrays. The insertion function inserts subflow records to bucket arrays. Finally, the query function enables the query over network flows.

In the following, we first present ingestion and membership functions for subflow streams, and then present strawman approaches based on popular sketch methods. Next, we present the insertion and query operations. Finally, we present sketching applications and the performance analysis.

B. Subflow Ingestion and Membership

1) Subflow Stream: As long as a subflow's accumulated counter exceeds a pre-defined threshold τ (128 by default) or a maximal waiting period expires, we evict this subflow immediately from the hash table and put it to a message bus. As a result, the subflow record's counter is at most τ. The threshold τ is adjustable to account for the distribution.

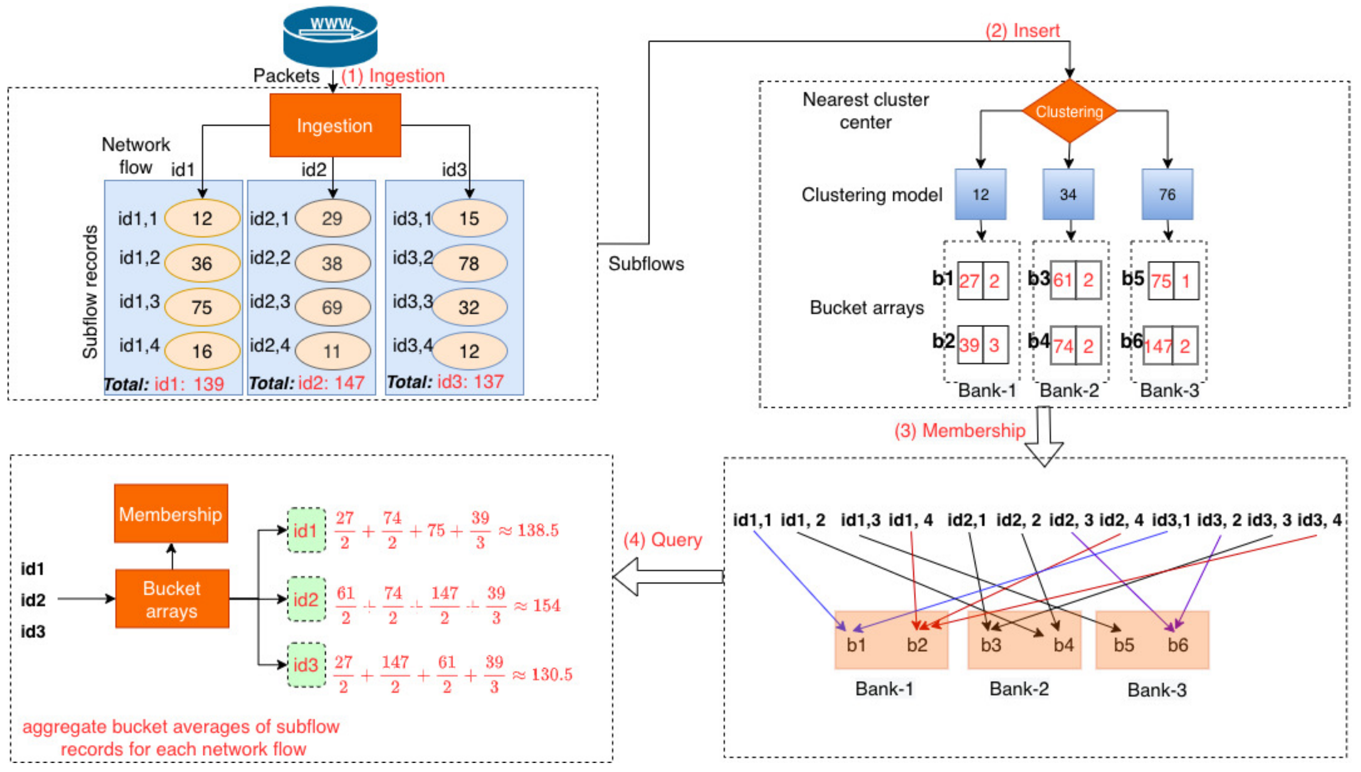


Fig. 2. Overall framework of Jellyfish. Step (1): Three network flows denoted as id1, id2 and id3 are divided to subflow records, each of which consists of four subflow records. The sizes of network flows denoted as id1, id2 and id3 equal the sums of the corresponding subflow records belonged to the same network flow, which are 139, 147 and 137, respectively. Step (2): Subflow records are clustered based on a clustering model (12, 34, 76). Each subflow record is mapped to the bucket array indexed by the nearest cluster center. Next, a bucket is chosen from this bucket array based on the hash of the subflow identifier with a hash function. Finally, the *sum* field of this bucket is increased by the incoming subflow record, and the *count* field of this bucket is incremented by one. Step (3): The membership of subflows are kept for query purpose. Step (4): Three network flows are estimated based on the sum of queried results of the corresponding subflow records. The estimation results for id1, id2 and id3 are 138.5, 154 and 130.5, respectively.

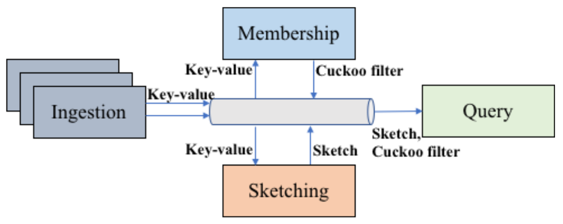


Fig. 3. Key functions in a subflow based network-monitoring architecture.

Due to the randomness of the ingestion function, multiple records may have the same network-flow identifier, since large network flows are captured in different records by the ingestion function.

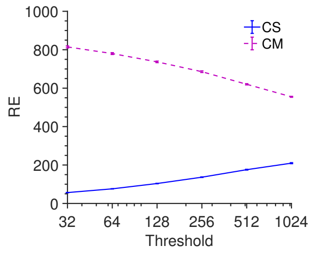
2) Subflow Membership: The membership function extracts the network-flow identifiers from the ingestion functions and stores them to a persistent membership data structure.

To avoid keeping an in-memory cache to remember each network flow, we account for each subflow record with a simple schema. Specifically, we denote each subflow record as a distinct subflow record, by appending a monotonically-ascending index to the original network-flow record. Thus, each subflow record now has a distinct representation: for a network-flow identifier x, the subflow identifier is of the form: recordID(x) = "(x, Index(x))", where Index(x) returns a monotonically-increasing index for the identifier x.

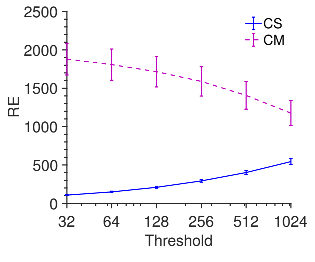
First, we approximately keep the set of subflow records by attaching a Cuckoo filter to each bucket array. The Cuckoo filter inserts subflow records based on cuckoo hashing, supports efficient insertion and deletion of items, and is shown to be more efficient than the Bloom filter at low false positives [22], [23], [24]. Multiple subflow records may be mapped to the same slot in the Cuckoo filter, as we would like to reduce the storage cost at the expense of introducing a degree of false positives.

Second, we track the cardinality of subflow records based on a Cuckoo filter. This *subflow tracker* keeps an integer-numbered counter f_s, which monotonically counts the number of subflow records for the same network flow x, i.e., the cardinality of the subflow records. Suppose that the cardinality field f_s amounts to three for a network-flow x, we immediately know that we have inserted three records to the Jellyfish with keys "(x, 1)", "(x, 2)", "(x, 3)". Thus f_s increases by one for each new record. The cardinality field f_s just adds four-bytes storage to the entry.

The storage of subflow membership may use more space-optimized data structures, such as the count-min sketch, at the cost of introducing a degree of approximation errors for the



(a) CAIDA



(b) MAWI

Fig. 4. The average and the 95-th confidence intervals of the relative errors for the per-flow query as we change the threshold τ for CS and CM methods. The sketch is of size 5 KB. The x-axis is plotted in the logarithmic scale.

subflow membership.

C. Limitations of Strawman Approach

Having presented the subflow stream, a strawman approach to improve the prediction accuracy would be to map subflow records to the sketch and aggregate the subflow estimations to reconstruct the network-flow level estimators. In this subsection, we illustrate with a real example why this approach does not work.

We test this hypothesis with two well-known sketch methods count-min (CM) [15] or count-sketch (CS) [16]. Figure 4 plots the average relative error as a function of the subflow threshold τ for CS and CM on two real-world data sets introduced in Subsection V-A. Compared to Figure 5 without the subflow stream, we see that this strawman approach does not decrement the relative errors for CS and CM, since peak subflow counters still increment the relative errors under hash collisions, and the sum of subflow estimations further amplifies the overall estimation. Further, the relative error of the CS sketch increases quickly as we increment the threshold τ. Since CS chooses the median of the estimation for each subflow record, which are sensitive to enlarging peak subflow counters. While the relative error of the CM sketch decreases with increasing subflow thresholds, since a CM sketch always chooses the minimum of the estimation for each subflow record, which is less sensitive to enlarging thresholds.

In summary, the proposed strawman approach does not improve the query performance due to the hash collisions of peak subflow counters. We next present a subflow clustering based sketch to resolve this issue.

D. Jellyfish Sketch

1) Physical Structure: The storage of Jellyfish is comprised of a number k of bucket arrays for subflow clustering. A bucket array consists of a number of buckets, where each bucket has two fields: (i) A *sum* field that records the sum of subflow values; (ii) A count filed that records the number of subflow records inserted to this bucket.

2) Insert: Jellyfish maps each subflow record (x, v) to the nearest cluster center with respect to v. We present the insertion process in Algorithm 1. We select the bucket array corresponding to the cluster index of the incoming record x, then we find a bucket in this bucket array with the hash of the key κ = recordID(x), and increment the counter of the mapped bucket by the hash of the key κ with the subflow counter v: *sum* = *sum* + v, and *count* = *count* + 1.

Algorithm 1: Insert operation on the Jellyfish.

```
1 Insert($x$, $v$)
  input : Network-flow identifier $x$, subflow counter $v$.
2 Index($x$) ← the cardinality field $f_s$ of the key $x$ in the subflow
  tracker, incrementing $f_s$ by one;
3 Generate the key $\kappa$ = ($x$, Index($x$));
4 Find the bucket array index $i_\kappa = argmin_i \|v - \mu_i\|$ to the
  nearest cluster center;
5 Update the bucket indexed at $h(\kappa)$: $I_{i_\kappa}[h(\kappa)].sum+=v$,
  and $I_{i_\kappa}[h(\kappa)].count$ +=1;
6 Insert $\kappa$ to the $i_\kappa$-th Cuckoo filter;
```

Algorithm 2: Query operation on the Jellyfish.

```
1 Query($x$)
  input : Network-flow identifier $x$.
  output: Network-flow counter $v$.
2 Obtain the cardinality field $f_s$ (denoted as $x_s$) of the key $x$
  from the subflow tracker;
3 SumCounter = 0;
4 for $j = 1 \rightarrow x_s$ do
5     Generate the key $\kappa$ = ($x$, $j$);
6     Obtain the bucket-array index $i_\kappa$ for $\kappa$ in the bank of
      Cuckoo filters;
7     SumCounter + = $\frac{I_{i_\kappa}[h(\kappa)].sum}{I_{i_\kappa}[h(\kappa)].count}$;
8 return SumCounter;
```

Complexity: During the insertion phase, Jellyfish avoids the maintenance of the volatile network-flow counters in memory, and does not need to adjust the mapping between network flows and the clustering model as well as the bucket arrays. The downside of these benefits, is that Jellyfish needs to save the subflow membership instead of the network-flow membership. We need one Cuckoo query to obtain the cardinality of the subflow records f_s and increment it by one, then we need one hashing evaluation to locate a bucket in a single bucket array. Finally, we need to insert the subflow key to the corresponding Cuckoo filter associated with this bucket array for the approximate membership.

3) Query: We next present the query process in Algorithm 2. For a network-flow identifier x, we first query the cardinality field f_s for key x from the Cuckoo filters, then we construct the set of keys $KEYs(x)$: "(x, 1)", "(x, 2)", …, "(x, f_s)". Next, for each key y in $KEYs(x)$, we locate the index of the bucket array by querying the Cuckoo filter associated with each bucket array. Next, we select the bucket in the corresponding bucket array with the hash of the key, and return the division $\frac{sum}{count}$ as the approximate counter for this key y. Finally, we calculate the sum of approximate counters for each key in $KEYs(x)$, and return this number as the counter for the network flow x.

Complexity: During the query phase, we need one Cuckoo query to obtain the cardinality field f_s of this network flow, then we obtain the index for each of the key by querying the bank of the Cuckoo filters. The query time is proportional to the cardinality number of subflow records. As the query phase is not in the critical path of the network-monitoring application, the speed of the query phase is much less important than that of the insertion phase.

E. Sketching Applications

The query function performs monitoring queries on sketches from the sketching function with network-flow identifiers from the membership function. Typical queries include: (a) **Per-flow frequency and entropy query**. They track the traffic volume of each distinct flow, or count the flow bytes. To query the size distribution of each inserted flow, we iteratively obtain approximation results with identifiers of inserted flows, then we build a list of approximated flow sizes as the flow size distribution. Similarly, we derive the entropy metric as the frequency distribution of approximated flow sizes. (b) **Heavy hitters**. It finds top-K flows ingesting the most traffic volumes. For a given heavy-hitter detection threshold, we obtain approximated values of inserted flows from the sketch, and select those exceeding the threshold as heavy hitters. Based on heavy hitters, we can also find flows spanning multiple windows that fluctuate beyond a predefined threshold, i.e., the heavy changes.

F. Theoretical Analysis

Having presented the Jellyfish method, we next bound the accuracy of Jellyfish towards the expectation of the ground-truth counter of each network flow. As shown in Theorem 1, the estimated counter just deviates the expectation of the ground-truth counter within a short interval, which is proportional to the number of subflow records of each network flow.

Theorem 1. *Suppose that a sequence of independently identically distribution (iid) subflow records are inserted to a bucket array with m buckets, with expectation ϕ and variance σ^2. Let the estimated counters of a list of network-flow identifiers S be represented as X. Assume that the l-th key is mapped to the j-th bucket. Let $Query(X_l)$ denote the estimator of the ground-truth counters X_l for the l-th network flow in S. Let $f_s(X_l)$ denote the cardinality number of subflow records for network-flow X_l. Then the Jellyfish estimator satisfies that*

$$Pr\left(|Query(X_l) - E\left[X_l\right]| \geq f_s(X_l)w\right) \leq \frac{\tau^2}{w^2}, w > 0 \quad (1)$$

Proof. Let n_j denote the number of subflow records inserted to the j-th bucket. For a bucket j, let the subflow records mapped to this bucket be represented as $\left\{Z^{(j)}\right\}$. Let $Y_j = \frac{\sum_z Z^{(j)}(z)}{n_j}$ denote the average of the j-th bucket.

(i) Expectation. Due to the *iid* distribution of subflow records, the expectation of the variable Z satisfies that $E\left[Z\right] = \phi$, thus the expectation of the variable Y_j amounts to the expectation of the variables:

$$E\left[Y_j\right] = \frac{1}{n_j} E\left[\sum_i Z^{(j)}\right] = \frac{1}{n_j}\sum_i E\left[Z^{(j)}\right] = \phi \quad (2)$$

(ii) Variance. Next, the variance of Y_j from its expectation can be calculated as follows:
$Var\left[Y_j\right] = E\left[\left(Y_j - \phi\right)^2\right] = E\left[\left(\frac{\sum_i Z^{(j)}}{n_j} - \phi\right)^2\right] \leq$
$E\left[\frac{1}{{n_j}^2}\left(\sum_i\left(Z^{(j)} - \phi\right)\right)^2\right] \leq \frac{1}{{n_j}^2} \times n_j^2\tau^2 = \tau^2,$
since the difference of subflow records to the average ϕ is always bounded by the subflow threshold τ for any subflow key z.

(iii) Bound. By Chebyshev's inequality, we bound the range of Y_j as :

$$Pr\left(|Y_j - \phi| \geq w\right) \leq \frac{Var\left[Y_j\right]}{w^2} \leq \frac{\tau^2}{w^2} \quad (3)$$

(iv) Network flow. Given any network-flow identifier X_l, we see that the expectation of X_l can be represented as the sum of counters with the same network-flow identifier: $E\left[X_l\right] = E\left[\sum_{FlowID(z)=X_l} Z^{(j)}(z)\right]$, where $FlowID(z)$ denotes the network-flow identifier of the subflow record z. We see that $E\left[X_l\right] = \sum_{FlowID(z)=X_l}\phi = f_s(X_l)\phi$.
Thus we have
$Pr\left(|Query(X_l) - E\left[X_l\right]| \geq f_s(X_l)w\right) =$
$Pr\left(|Query(X_l) - f_s(X_l)\phi| \geq f_s(X_l)w\right) =$
$Pr\left(|Y_j \cdot f_s(X_l) - f_s(X_l)\phi| \geq f_s(X_l)w\right) =$
$Pr\left(|Y_j - \phi| \geq w\right)$ (dividing constant $f_s(X_l)$ at both sides)
$\leq \frac{\tau^2}{w^2}$(Eq.(3)).

□

IV. Implementation

We implement the Jellyfish sketch in modular blocks based on a publish/subscribe (Pub/Sub for short) framework.

The Pub/Sub framework provides seamless messaging supports for monitoring functions. One or multiple producer entities publish messages towards the same topic, then the Pub/Sub messaging framework delivers ordered messages to consumers subscribed to the same topic.

The ingestion function aggregates packets at line rates to subflow records [25], and evicts subflow record messages by predefined threshold τ. The hash table reduces packet-processing delay by cache prefetching and batch processing.

The online subflow clustering procedure aims to reduce the variance of subflows in each bucket array. We initialize the clustering model with a set of subflow samples. Next, we adapt the clustering model to be aware of the variations of subflow distributions. It is not necessary to strictly adhere to the subflow distribution, since an approximate clustering model still reduce the variance compared to existing sketching methods. We maintain the subflow clustering model based the K-means clustering method that consists of k cluster centers. The number k controls the number of clusters. The distance from the item to a cluster center is defined as the one-dimensional absolute difference between the subflow record and the cluster center. Let S denote the set of one-dimensional samples. Let the distance measure between two sample x and y ($x, y \in S$) be the absolute difference $|x - y|$. The clustering model finds a set of k points denoted as μ to minimize the variance of values within each cluster.

V. Evaluation

Having presented the Jellyfish method, we next report experiment results compared to state-of-the-art methods with real-world data sets.

A. Experimental Setup

Data sets: We perform a real-world trace-driven experiment study with two popular data sets: (i) *CAIDA*: it is collected on February 18, 2016 at the Equinix-Chicago monitor by CAIDA [12], with 1799.7 million stream network flows lasting for one hour. (ii) *MAWI*: it is collected on May 20, 2019 at the transit link of WIDE to the upstream ISP [26], with 14.0 million stream network flows lasting for 899.99 seconds. The source IP of each stream network flow serves as the network flow's key.

Workflow: We split each data set to ten intervals of equal size. Each interval is replayed to the ingestion function. The ingestion function publishes subflow tuples to sketching components over the Pub/Sub framework. This sketch is queried by the operator after the interval ends.

Metrics: We evaluate the effectiveness of the sketch methods with three metrics: (i) *Relative error* (RE): We use the relative error to quantify the accuracy of the per-flow query. It is defined as the mean of the relative error of each queried network flow. (ii) *F1 score*: We use the F1 score to quantify the precision for the heavy-hitter query. It is defined as the harmonic mean of the precision and the recall values, i.e., $\frac{2\mathrm{PR}\times\mathrm{RR}}{\mathrm{PR}+\mathrm{RR}}$, where PR (Precision Rate) denotes the percent of true heavy-hitter instances reported, and RR (Recall Rate) denotes the percent of found true heavy-hitter instances.

Parameters: We select the default parameters for Jellyfish based on the sensitivity experiments. We set the default number of clusters to 30, and the default number of buckets in Jellyfish to 0.1 times the number of network flows in an evaluation interval. We set the truncated threshold for subflow tuples to 128.

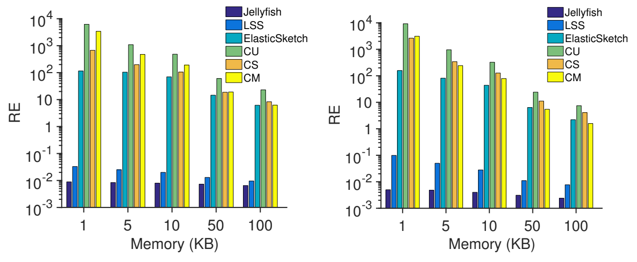
The experiments are repeated in ten times, and we report the average result and the 95-th confidence interval.

B. Comparison

We first compare Jellyfish with state-of-the-art sketching methods using the same space for fair comparison. We report average metrics as a function of the total amount of memory of the sketch physical structure. We keep the membership of inserted keys with the same set of Cuckoo filters, since all sketching methods are agnostic of the keys by themselves.

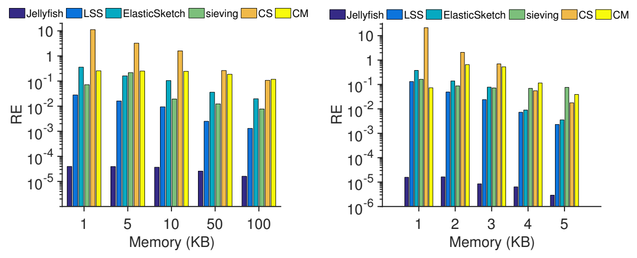
1) Per-flow Query: We first test the relative errors of the per-flow query. We compare Jellyfish with five per-flow sketching methods including count-sketch (CS) [16], cusketch (CU) [1], count-min (CM) [15], Elastic Sketch (ES) [12] and LSS [14].

Figure 5 shows the relative errors as we vary the bucket storage from one KB to 100 KB. We see that the relative errors of Jellyfish and LSS are three to five orders of magnitude smaller than those of CS, CM, CU and ElasticSketch. Further, Jellyfish is significantly more accurate than LSS, since the record distributions in Jellyfish are less skewed than those in LSS.



(a) CAIDA (b) MAWI

Fig. 5. The average of the relative errors for the per-flow query. The y-axis is plotted in the logarithmic scale.



(a) CAIDA (b) MAWI

Fig. 6. The average of the relative errors for the entropy query. The y-axis is plotted in the logarithmic scale.

2) Entropy Query: We next compare the relative errors of the entropy query of Jellyfish with five entropy-sketching methods including count-min (CM) [15], count-sketch (CS) [16], Sieving [27], Elastic Sketch (ES) [12] and LSS [14].

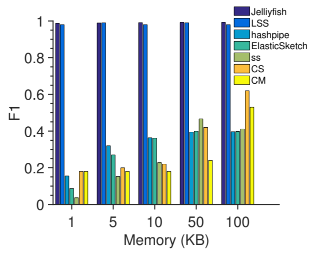
Figure 6 shows the relative errors as we vary the sketching storage from one KB to 100 KB. We see that Jellyfish is three to six orders of magnitude more accurate than the other methods. This is due to the fact that Jellyfish clusters less-skewed subflow records to bucket arrays, which closely preserves the global distribution of the network-flow counters.

3) Heavy Hitter: We next compare the F1 scores of the heavy-hitter query with six heavy-hitter methods including count-min (CM), count-sketch (CS), Spacesaving (SS) [2], ElasticSketch (ES), and hashpipe [4] and LSS [14]. We set the threshold of the heavy hitters to the top-5% of network flows.

Figure 7 shows the F1 scores as we increase the sketching storage from one KB to 100 KB. We see that the F1 scores of both Jellyfish and LSS are close to one, since the heavy hitters depend on a small set of the largest network flows, and both methods put large items to the same bucket arrays by the clustering process. While hashpipe, ElasticSketch, SS, CS and CM are more sensitive to the mixing of small items with large items.

C. Sensitivity

Having shown that Jellyfish outperforms state-of-the-art methods for different query requests, we next evaluate the sensitivity of Jellyfish by varying the choice of parameters. The


(a) CAIDA

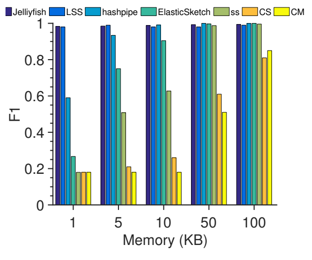
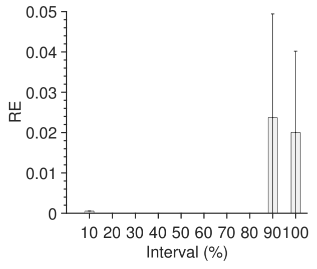
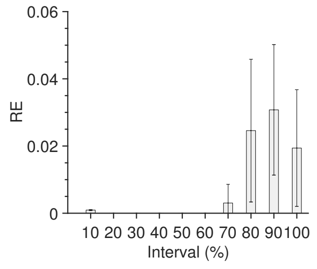
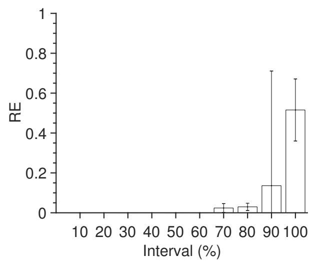

(b) MAWI

Fig. 7. The average of the F1 score of compared methods.


(a) CAIDA, Jellyfish


(b) MAWI, Jellyfish


(c) CAIDA, LSS

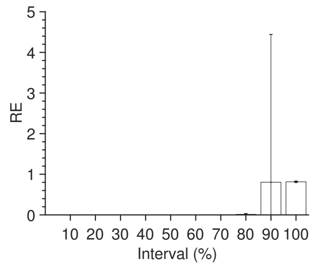
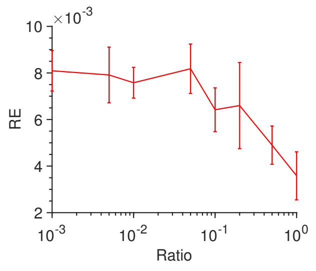

(d) MAWI, LSS

Fig. 8. The average and the 95-th confidence interval of relative errors for partitioned network-flow ranges for Jellyfish and LSS with 5 KB-sized bucket arrays.


(a) CAIDA

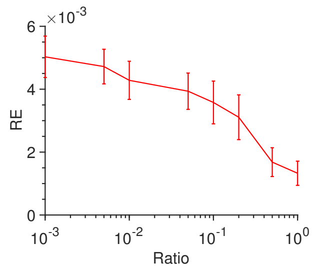
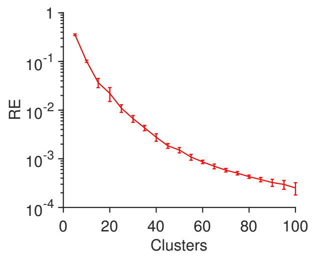

(b) MAWI

Fig. 9. The average and the 95-th confidence interval of relative errors as a function of the ratio of the number of buckets to the number of network flows in an interval. The x-axis is shown in logarithmic scale.


(a) CAIDA

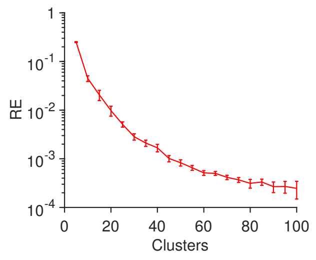

(b) MAWI

Fig. 10. The average and the 95-th confidence interval of relative errors as a function of the number of clusters.

average ingestion rate for a single-thread ingestion function reaches 22.35 million packets per second on the Intel Core i7 CPU with Quad-core.

For each group of experiments, we fix all but one parameters and show the variations of the relative errors for the per-flow query, since this query captures the fine-grained query performance. We follow the default parameters for Jellyfish presented in subsection V-A.

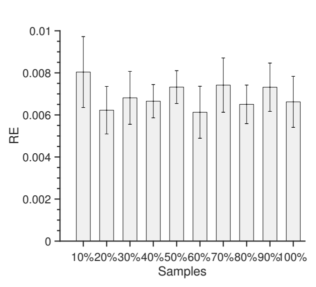
1) Range: We first report the distributions of per-flow errors for network-flow counters in different ranges. We partition the network-flow counters to ten equal-sized groups separated by the 10-th, 20-th to 90-th percentiles in the distribution. We plot the average and the 95-th confidence intervals of the per-flow metric for each network flow counter, and group them by the group of network-flow counters.

Figure 8 plots the per-flow relative error for each group of network-flow counters of Jellyfish and LSS. We see that the per-flow errors are mainly caused by top network-flow counters that are located in the last two to three groups, i.e., the tail of the network flow distribution, while the relative errors of the rest of groups are zeros or close to zeros. Moreover, Jellyfish's per-flow errors for the tail distribution are over ten times smaller than those of LSS, thanks to the clustered subflow records to bucket arrays.

2) Bucket Ratio: We next vary the ratio of the number of buckets in the Jellyfish to the number of network flows in an measurement interval. Figure 9 shows the average and the 95-th confidence intervals of the per-flow relative error as a function of the ratios ranging from 0.001 to 1. We see that the per-flow relative errors of Jellyfish at the bucket ratio 0.001 are around 0.008 and 0.005 for the CAIDA and MAWI data sets, respectively. Incrementing the bucket ratio from 0.001 to 1 reduces the average relative error by one to two times. Thus, Jellyfish is reasonably accurate under severe hash collisions.

3) Number of Clusters: We next vary the number of clusters and see the variations of per-flow query performance. Figure 10 plots the average and the 95-th confidence interval of the per-flow relative error as a function of the number of clusters. We see that the per-flow relative error decrements quickly as the number of clusters increments from 5 to 40, and reach the diminishing returns with around 60 clusters. Increasing the number of clusters leads to a fine-grained partitioned subflow clustering model, which separates more dissimilar subflow records to different bucket arrays accordingly.

4) Clustering Samples: We next test the clustering stability as we change the number of clustering samples for the K-means clustering training process. We vary the number of cluster samples as a function of the percent of the number of subflow records in an interval. We plot the average and the 95-th confidence interval for the relative-error metric. Figure 11


(a) CAIDA

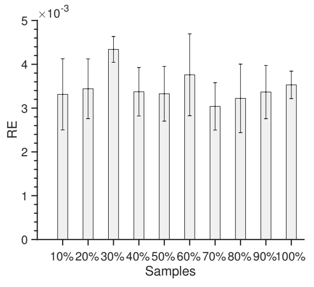
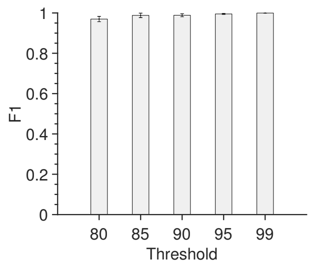

(b) MAWI

Fig. 11. The average and the 95-th confidence interval of relative errors as a function of the percent of cluster samples.


(a) CAIDA

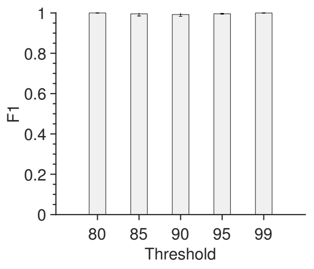
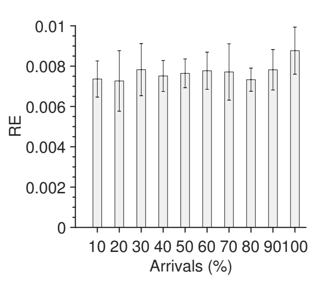

(b) MAWI

Fig. 12. The F1 scores as a function of the threshold of heavy hitters.


(a) CAIDA

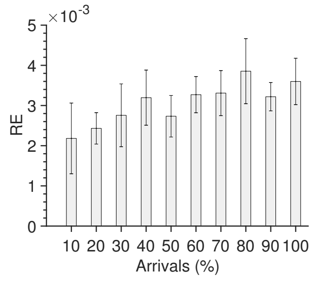
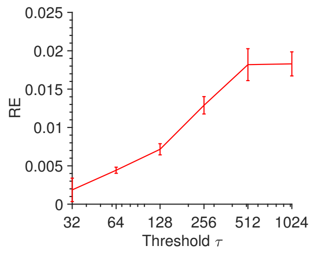

(b) MAWI

Fig. 13. The average and the 95-th confidence interval of relative errors as a function of the percent of arrived network flows.


(a) CAIDA

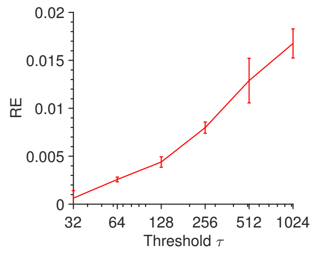
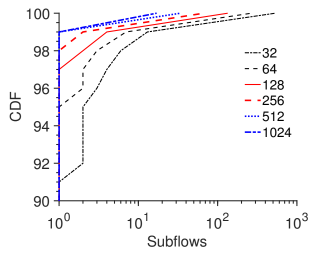

(b) MAWI

Fig. 14. The average and the 95-th confidence interval of relative errors as a function of the thresholds.


(a) CAIDA

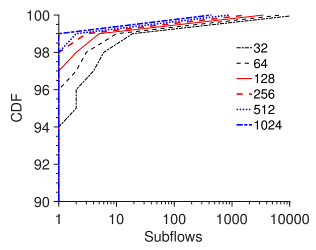

(b) MAWI

Fig. 15. The cumulative distribution function (CDF) of the numbers of subflows per network flow. The x-axis is plotted in logarithmic scale.

plots the variations of per-flow relative errors as we change the number of cluster samples from 10% to 100% of the number of records in an interval. We see that the per-flow relative errors remain fairly stable. Thus a small number of clustering samples are enough to obtain fairly accurate results.

5) Heavy Hitter: We next test the per-flow relative errors as we vary the heavy-hitters from those greater than 80% to 99% of network flows. Figure 12 shows the average and the 95-th confidence of the F1 scores as we change the thresholds of heavy hitters. We see that the F1 scores are more close to one with increases thresholds towards 99%. Thus Jellyfish remains stably accurate for heavy-hitter queries for a wide range of parameters.

6) Dynamics: We next plot the variations of the per-flow relative errors as we gradually insert items to a fixed Jellyfish sketch. We plot the average and the 95-th confidence interval for the relative-error metric. Figure 13 shows that the per-flow relative errors increase gracefully with increasing records from 10% to 100%. This is because Jellyfish clusters similar subflow records to the same bucket array, so that the estimation is less affected by hash collisions of subflows.

7) Subflow Threshold τ: We next plot the variations of the per-flow relative errors as we change the subflow threshold τ. We plot the average and the 95-th confidence interval for the relative-error metric. Figure 14 shows that the per-flow relative errors increases gracefully with increasing thresholds from 32 to 1024. This is because larger subflow thresholds increases the chance to put more dissimilar subflow records to the same buckets. On the other hand, reducing the subflow thresholds creates a higher-rate stream of subflow records. As a result, we should choose a modest subflow threshold to trade off the accuracy and the subflow rate.

We next count the number of subflows per network flow for different subflow thresholds τ. Figure 15 shows the cumulative distribution functions of the numbers of subflows. We see that over 90% of network flows only have one subflow tuple, since the majority of network flows are small flows. Further, as we increase the subflow threshold τ from 32 to 128, 99% of the numbers of subflows decrements from 13 to 4 for the CAIDA data set, and decrements from 19 to 5 for the MAWI data set, respectively.

VI. Conclusion

We present a locality-sensitive subflow sketching method Jellyfish that disaggregates a single network-flow counter to multiple subflow counters, and inserts similar subflow counters to the same bucket array based on subflow-clustering model. Real-world data sets based experiments show that Jellyfish significantly reduces the per-flow query errors by orders of magnitude with low variance within bucket arrays. Jellyfish is reasonably accurate for a wide range of parameters.

The implementation will be made publicly available. As our future work, we plan to find more space-optimized data structures to track subflow membership. It would be interesting to replace Cuckoo filters with more space-efficient in-memory data structures [28], [29].

REFERENCES

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

[2] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, 2005, pp. 398–412.

[3] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, 2008, pp. 121–132.

[4] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, 2017, pp. 164–176.

[5] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002, pp. 346–357.

[6] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*, 2003, pp. 234–247.

[7] A. Kumar, M. Sung, J. J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, 2004, pp. 177–188.

[8] M. Yoon, T. Li, S. Chen, and J. Peir, "Fit a compact spread estimator in small high-speed memory," *IEEE/ACM Trans. Netw.*, vol. 19, no. 5, pp. 1253–1264, 2011.

[9] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 101–114.

[10] G. Cormode, "Data sketching," *Commun. ACM*, vol. 60, no. 9, pp. 48–55, 2017.

[11] R. Ben-Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017, pp. 127–140.

[12] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 561–575.

[13] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 576–590.

[14] Y. Fu, D. Li, S. Shen, Y. Zhang, and K. Chen, "Clustering-preserving network flow sketching," in *IEEE INFOCOM*, 2020.

[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[16] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, 2002, pp. 693–703.

[17] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017, pp. 113–126.

[18] G. Cormode and M. Hadjieleftheriou, "Finding the frequent items in streams of data," *Commun. ACM*, vol. 52, no. 10, pp. 97–105, 2009.

[19] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, J. Wu and W. Hall, Eds. ACM, 2019, pp. 334–350.

[20] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *ACM SIGCOMM*, 2020.

[21] CAIDA, "Trace statistics for caida passive oc48 and oc192 traces," http://www.caida.org/data/passive/trace_stats/, 2018.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, 2014, pp. 75–88.

[23] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, 2013, pp. 97–108.

[24] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy bloom filters for multi-set membership testing," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, 2016, pp. 139–151.

[25] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 129–143.

[26] W. M. WorkingGroup, "Packet traces from wide backbone," http://mawi.wide.ad.jp/mawi/, 2019.

[27] A. Lall, V. Sekar, M. Ogihara, J. J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006*, 2006, pp. 145–156.

[28] Y. Zhang, D. Li, Z. Sun, F. Zhao, J. Su, and X. Lu, "CSR: classified source routing in distributed networks," *IEEE Trans. Cloud Comput.*, vol. 6, no. 2, pp. 464–477, 2018. [Online]. Available: https://doi.org/10.1109/TCC.2015.2440242

[29] Y. Zhang, D. Li, C. Guo, H. Wu, Y. Xiong, and X. Lu, "Cubicring: Exploiting network proximity for distributed in-memory key-value store," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2040–2053, 2017. [Online]. Available: https://doi.org/10.1109/TNET.2017.2669215