Short-Term Memory Sampling for Spread Measurement in High-Speed Networks

Yang Du[†], He Huang[†], Yu-E Sun[‡], Shigang Chen[§], Guoju Gao[†], Xiaoyu Wang[†], Shenghui Xu[†]
[†]School of Computer Science and Technology, Soochow University, Suzhou, China
[‡]School of Rail Transportation, Soochow University, Suzhou, China
[§]Department of Computer and Information of Science and Engineering, University of Florida, US

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796702

***Abstract*—Per-flow spread measurement in high-speed networks can provide indispensable information to many practical applications. However, it is challenging to measure millions of flows at line speed because on-chip memory modules cannot simultaneously provide large capacity and large bandwidth. The prior studies address this mismatch by entirely using on-chip compact data structures or utilizing off-chip space to assist limited on-chip memory. Nevertheless, their on-chip data structures record massive transient elements, each of which only appears in a short time interval in a long-period measurement task, and thus waste significant on-chip space. This paper presents short-term memory sampling, a novel spread estimator that samples new elements while only holding elements for short periods. Our estimator can work with tiny on-chip space and provide accurate estimations for online queries. The key of our design is a short-term memory duplicate filter that reports new elements and filters duplicates effectively while allowing incoming elements to override the stale elements to reduce on-chip memory usage. We implement our approach on a NetFPGA-equipped prototype. Experimental results based on real Internet traces show that, compared to the state-of-the-art, short-term memory sampling reduces up to 99% of on-chip memory usage when providing the same probabilistic assurance on spread-estimation error.**

Index Terms*—Traffic measurement, spread estimation, short-term memory sampling.

I. Introduction

Nowadays, with the dramatic increase in network size and transmission speed, robust network measurement in high-speed networks has become an indispensable function for service providers and network administrators [1]–[8]. This paper focuses on per-flow spread measurement (*i.e.*, counting the number of *distinct* elements) [9]–[13], which is a particularly challenging measurement task compared to size measurement (*e.g.*, counting the number of packets) because it needs to identify the unique elements and the duplicates. The flows under measurement can be TCP flows, HTTP flows, or any sequence of packets identified by the same flow label, *e.g.*, source/destination address. The elements can also be flexibly defined as the address, port, or other header fields to meet the application requirements. For instance, we can define the flow label as the source address, let the element be the destination address, and identify the scan attackers as the sources with abnormal flow spreads, *i.e.*, reporting the sources that have contacted with an abnormal amount of hosts [14]–[17].

However, it is challenging to implement an online per-flow spread measurement function module to process packet streams at the line rate of modern routers (such as 100Gbps and beyond) while providing accurate measurement results for millions of flows [18]. The primary challenge is the constraints for memory resources. Per-flow spread measurement demands large memory capacity and large memory bandwidth for element recording and duplicate filtration, but off-the-shelf memory modules cannot achieve high-speed and large-capacity simultaneously [19]–[21]. More specifically, on-chip memory (SRAM) is fast enough to support line-speed read/write operations but only has a small capacity (usually less than 10MB [20]) which is insufficient for precise measurement. The off-chip memory (DRAM) is large enough to hold all packets, while its update speed is much slower than the packet forwarding rate.

Many approaches have been proposed to address the above challenges, which can be roughly categorized into *sketch-based* [22]–[26] and *sampling-based* [27]–[29]. The former chooses to place their compact data structures (*i.e.*, sketches) solely in the on-chip memory to catch up with line speed. However, sketches achieve high space efficiency by sharing the recording units (bits/registers) among flows, which inevitably decreases the measurement accuracy. Moreover, sketch-based methods are only suitable for offline queries since it is time-consuming to scan hundreds or thousands of units to provide spread estimation for each flow. The sampling-based approaches are proposed to complement the sketch-based ones. Typically, they use an on-chip sampling part to filter the duplicates and sample the non-duplicates (*i.e.*, unique elements) at a well-designed sampling rate. The selected elements will be offloaded to the off-chip recording module and stored in separate records assigned for each flow to reduce estimation errors and support online queries.

However, these prior studies neglect that network traffic is mostly transient and bursty, *e.g.*, more than 60% of elements (destination addresses) in the one-minute Internet trace downloaded from CAIDA [30] are transient elements that appear for less than 2 seconds. Consequently, their model choice of placing long-term memory data structures on the chip will waste most on-chip space by persistently recording all elements, especially the massive transient elements that only last for short durations. Moreover, in case of setting a longer measurement period or meeting a sudden traffic flood, these approaches will demand much more on-chip space to record the increasing number of elements.

We aim to complement the prior studies by adequately

978-1-6654-5822-1/22/$31.00 ©2022 IEEE

dealing with transient network traffic, which is challenging and has not been investigated before. An intuitive thought is removing transient elements right after their last appearances, which is, however, infeasible since we cannot identify if an element is a transient one nor recognize the last appearance.

In this paper, we introduce a novel spread measurement solution, which is sampling new elements with a *short-term memory duplicate filter*. Notice that a *short-term memory duplicate filter* differs from long-term memory approaches in that it only keeps the recently appeared elements on-chip for a short period and likely forgets the stale elements. Thus, the oblivion of stale elements can make room for incoming elements, which is the key to measuring per-flow spread with minimal on-chip memory usage. Suppose that a transient element has appeared multiple times in a short period. Our solution will sample its first appearance with a pre-defined probability, then accurately identify its subsequent appearances as duplicates since short-term memory is accurate when the time interval is short. After that, when this element becomes stale, *e.g.*, the last appearance occurred ten seconds ago, our solution will forget this element with an oblivion probability that grows as time goes by. Therefore, most elements will not persist in our data structure, implying that we can efficiently measure per-flow spreads with tiny on-chip space.

Nevertheless, it is tricky to design a short-term memory data structure to record the element for short durations. The most related work is time-decaying Bloom Filters and their variants [31]–[33]. However, they are not suitable for per-flow spread measurement since they either require multiple memory accesses per packet or assume that the elements in the stream follow a uniform distribution. Further, short-term memory will introduce randomness to duplicate filtration and element sampling, raising new challenges for spread measurement.

This paper presents short-term memory sampling (STMS), a novel per-flow spread estimator based on the on-chip/off-chip model [27]–[29]. An on-chip sampling module is utilized to record the elements using short-term memory and sample the new elements with a pre-defined probability. The key of the sampling module is a short-term memory duplicate filter (STM-DF) which filters the duplicates in the packet stream and reports the new elements to the sampling module, with only one memory access per packet. The sampled elements will be offloaded to the off-chip recording module, where each flow is assigned with a separate record structure in the off-chip memory. This on-chip/off-chip design allows us to answer online spread queries and provide performance guarantees for the measurement results with tiny on-chip memory usage. We implement our method on a NetFPGA-equipped prototype and run extensive experiments on real Internet traffic traces downloaded from CAIDA [30]. The experimental results show that our design can work with tiny on-chip memory (such as 1KB) to provide probabilistic assurance on the spread-estimation error, which outperforms the state-of-the-art by reducing up to 99% on-chip memory usage.

II. Preliminary

A. System model

We consider a per-flow spread measurement system that monitors the packet stream passing by the measurement point (*e.g.*, central router/gateway). For each incoming packet, the measurement module will first extract the flow label f and the element label e, which can be flexibly defined as the combinations of header fields to meet application-specific requirements. We view the packets with the same flow label as a flow and treat the packet stream as a set of flows. Our objective is to measure each flow f in terms of flow spread $\widehat{n_f}$, which refers to the number of distinct elements in a flow.

This paper adopts an on-chip/off-chip model [27], [28] for per-flow spread measurement. An on-chip sampling module is deployed on the network processor unit (NPU) to sample the new elements with a pre-defined probability. Its core is a short-term memory duplicate filter that reports new elements and filters the duplicates using an on-chip bit array (denoted as B). The elements selected by the sampling module will be offloaded to the off-chip recoding module, where each flow f is assigned with a separate set structure c_f in the off-chip memory. This model choice allows us to measure flow spreads with minimal on-chip memory usage and provide accurate estimation results for online queries in real-time.

B. Existing solutions and their limitations

Most existing spread measurement solutions are *sketch-based* [22]–[25], which use compact data structures (*i.e.*, sketches) to fit in limited on-chip space. However, this memory efficiency does not come for free. Sketches need to share the recording units (bits/registers) among flows, resulting in two common limitations. First, the choice of sharing the bits/registers will introduce noises to each flow's record and decrease the measurement accuracy, especially for the small flows. Second, when estimating the spread of a single flow, sketch-based solutions require scanning multiple hundreds or even thousands of bits/registers, which consumes significant time. In other words, sketch-based approaches are suitable for answering offline queries but do not support online queries.

Sampling-based methods [27]–[29] adopt an on-chip/off-chip model different from *sketch-based* solutions to utilize both on-chip and off-chip memory during measurement. They first employ an on-chip sampling module to sample the unique elements (non-duplicates) with the desired sampling probability. Then they store the sampled elements in the off-chip storage module to reduce estimation error and improve query throughput. However, to implement the above design, they place long-term memory duplicate filters on the chip to identify the unique elements. Since most network traffic is transient and bursty, this model will persistently hold the massive transient elements in the on-chip memory and significantly waste on-chip memory.

C. Requirements and challenges

We aim to fill the gap left by the prior art by addressing their limitations. Our goal is to design a short-term memory

sampling method that can effectively address the transient traffic and use minimal on-chip space to provide accurate estimations for online per-flow spread queries. There are three natural requirements that the proposed method should satisfy:

- **[R1]** *Short-term memory sampling:* It is inefficient to memorize all elements throughout the measurement period since most network traffic is transient and bursty. We want our design to achieve short-term memory sampling by filtering the duplicates with short-term memory and then sampling the unique elements with pre-defined probability.
- **[R2]** *Small and constant on-chip memory usage:* We want our method to use a small on-chip memory space to provide accurate spread estimation results. Also, we expect the required memory space to be constant and irrelevant to the measurement period, which offers strong guarantees on supporting different measurement applications and network environments.
- **[R3]** *Relative error bounds:* We want to provide a probabilistic guarantee for the relative errors of spread estimations with a form in [27]: Given a positive integer T, a relative error bound δ, and a probability threshold ϵ ($0 < \epsilon < 1$), the relative error of a flow with spread no smaller than T should be bounded by δ with a probability no less than $1-\epsilon$.

Given the trajectory of prior efforts that offer sketch-based measurement or non-duplicate sampling, implementing *[R1] short-term memory sampling* may appear infeasible. To achieve *[R2] small and constant on-chip memory usage*, we observe that when the network traffic volume grows, the memory space assigned for each flow will decrease, leading to great challenges in ensuring measurement accuracy. Finally, it is challenging to provide guarantees for *[R3] relative error bounds* with short-term memory sampling since the process of oblivion poses much uncertainty for the measurement results. In this paper, we present short-term memory sampling to satisfy the above requirements and overcome the challenges.

III. Overview of Short-Term Memory Sampling

A. Main idea

The main idea of short-term memory sampling is simple: *record the appeared elements with short-term memory and sample the new elements (identified by the short-term memory duplicate filter) with a pre-defined probability*. Notice that, unlike existing long-term memory estimators that hold every recorded element throughout the measurement, our short-term memory duplicate filter is expected to hold the elements only for short periods. Then the proposed duplicate filter can remember the recently appeared elements accurately but likely forget the stale ones. We believe this property is the key to properly dealing with transient traffic and achieving minimal on-chip memory usage.

We use an example to show the difference between the existing long-term memory sampling methods (*e.g.*, non-duplicate sampling) and our solution. Suppose that a transient element appears 1000 times in the first second of a 5-minute measurement task. Since short-term memory works well when the time interval is short, our method can sample this element's first arrival with a pre-defined probability, insert the element into the data structure, and then effectively remove the duplicates, which is identical to long-term memory approaches. The difference is that the long-term memory approaches will hold this element until the last second and result in a waste of on-chip space, but our method will only record this element for a short period. Therefore, when this element has not appeared for a long time, short-term memory will decay, and incoming elements will override the recorded information. With this property, we can reduce on-chip memory usage while guaranteeing measurement performance.

B. Architectural design

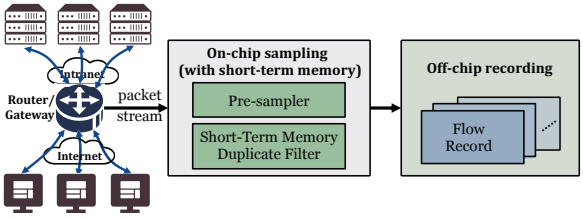


Fig. 1: Framework of short-term memory sampling

Fig. 1 depicts the structure of short-term memory sampling, which is built on an on-chip/off-chip model [27]–[29]. The proposed system contains two modules: an *on-chip sampling* module and an *off-chip recording* module. Their detailed descriptions are as follows:

The on-chip sampling module is deployed on the network processor unit (NPU) to process the packets at line speed. It employs a two-phase pipeline (*pre-sampler* and *short-term memory duplicate filter*) to achieve the design goal of short-term memory sampling. When receiving a packet stream, the pre-sampler first extracts elements from the packets and then samples and sends a pre-defined portion of elements to the duplicate filter. For each element sent from the pre-sampler, the duplicate filter will check whether it has seen the element before. If this is a new element, our filter will report it to the off-chip recording module. Notice that our short-term memory duplicate filter can achieve minimal memory usage. Thus, the on-chip sampling module can use tiny on-chip space to sample each distinct element with the same overall sampling probability, *i.e.*, each distinct element has the same probability of being offloaded to the off-chip recording module.

The off-chip recording module is often managed by a host server connected with the measurement point. It communicates with the on-chip part during the measurement period and records the elements sent from the on-chip part. With large-capacity off-chip memory, the recording module can assign each flow a separate record structure, eliminating the side effects of sharing and improving the measurement accuracy. When the measurement ends, the recording module can provide the flow records to the estimator to answer online per-flow spread queries.

With this on-chip/off-chip model, the proposed framework has two advantages: First, by using high-speed on-chip mem-

ory, it can process packet stream and sample elements at line rate, which guarantees a high processing throughput. Second, the off-chip recording module eliminates the noises caused by sharing and maintains more detailed information. Therefore, compared to the sketch-based approaches entirely based on on-chip memory, our framework can produce more accurate estimation results and support online queries.

C. Workflow design

In our design, the STMS framework supports two operations, *i.e.*, recording and estimation, to meet the requirement of online per-flow spread measurement.

1) Recording: For an incoming packet, the recording operation is conducted as follows: First, the router extracts flow label f and element label e from the packet and passes it to the on-chip sampling module. Then, the pre-sampler selects the element with a pre-defined probability. When an element is pre-sampled, it will be sent to the short-term memory duplicate filter to check whether this element is a duplicate. The non-duplicates reported by STM-DF will be sent to the off-chip recording module to update the flow record, *e.g.*, insert element e to a separate record structure c_f.

2) Estimation: When querying for the spread of an arbitrary flow f, the estimation operation is performed as follows: First, we retrieve the flow record from the off-chip recording module. If no record is found, STMS will regard this flow as an empty flow. Otherwise, STMS returns its estimation for the flow spread based on the flow record c_f and the overall sampling probability.

IV. Implementation of Short-term Memory Duplicate Filter (STM-DF)

A. Data structure

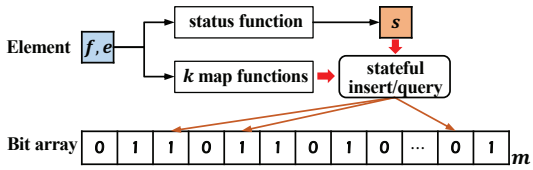


Fig. 2: Data structure of short-term memory duplicate filter

The data structure of STM-DF is depicted in Fig. 2, which contains a bit array B of m bits serving for the duplicate filtration. Meanwhile, STM-DF maintains k map functions $H_1(), H_2(), \cdots, H_k()$ and a status function $H_s()$. The map functions' output ranges are $\{0, 1, \cdots, m-1\}$ and they are designed to pseudo-randomly map each element to k bits of the bit array; the status function will randomly assign each element a status $s \in \{0, 1\}$.

With the above data structure, we introduce three novel stateful operations for STM-DF, which are the stateful initialization/insert/query, to filter duplicates with short-term memory (as illustrated in Section IV-C). The stateful initialization prepares STM-DF for further operations. The stateful insert writes an element's status into the bit array. In correspondence, the stateful query checks if an element's status has been recorded in the bit array.

We want to stress that the stateful operations we first propose for STM-DF are different from the traditional operations adopted by Bloom Filters [31], [32]: the traditional operations use k bits of '1' to indicate if an element exists, while our stateful operations allow each element to have a different status s ($\in \{0, 1\}$) and denote an element's existence by k bits of status s. Thus, when inserting an element e with status s into the filter, it naturally overrides the elements whose mapped bits overlap with e but have a different status. This design ensures that STM-DF only has a short-term memory since the probability that the incoming elements override a stale element will grow as the number of new elements increases. Moreover, compared to time-decaying Bloom Filters [31], [32] that require particular decay operations to remove the stale elements, our design of stateful operations can significantly reduce the memory access per packet to improve the processing throughput.

B. Stateful operations

Stateful initialization: The stateful initialization operation is conducted at the beginning of a measurement. It prepares STM-DF for further insert/query operations as follows: For each $B[i], i \in \{0, 1, \cdots, m-1\}$ in array B, we randomly initialize it to 0 or 1 with same probability. Thus, after initialization, all bits have a probability of 0.5 to be 0 and a probability of 0.5 to be 1, ensuring that the expected outcome from insert/query operations is identical for any element status s.

Stateful insert: The stateful insert operation takes an element (f, e) as input and insert its status information to the bit array, which is performed as follows: First, we run the status function $H_s(f \oplus e)$ to assign this element a status value $s \in \{0, 1\}$. Then we pseudo-randomly map this element to k bits of B through $h_i = H_i(f \oplus e), i \in \{1, 2, \cdots, k\}$. The insertion of element (f, e) is completed by setting the above k bits $(B[h_1], B[h_2], \cdots, B[h_k])$ to s.

For an arbitrary element, the k mapped bits have the same probabilities of being 1 or 0, and the element status is randomly chosen from $\{0, 1\}$. Therefore, we can ensure that the portions of ones and zeros in the bit array are both 50% during the measurement. This property promises that STM-DF is robust, and its duplicate filtration is irrelevant to time.

Stateful query: Similar to the stateful insert operation, the query operation also takes an element (f, e) as input. The difference is that it will check if this element exists in the STM-DF and returns a boolean value (*true* or *false*) to indicate the existence of the element. The detailed execution process is as follows: First, we assign this element a status $s = H_s(f \oplus e)$ and pseudo-randomly map this element to k bits in B. Then we consider two cases: The first case is that at least one bit out of the k bits has a value different from s, which means STM-DF has never seen this element before. In this case, STM-DF answers a *false* value for this query. The second case is that all the k bits have the same value of s. At this time, STM-DF will regard this element as one that has been inserted into the bit array and answer a *true* value for this query.

Due to the impossibility of achieving small memory usage and faultless duplicate filtration simultaneously, STM-DF, which only uses small on-chip space, sometimes returns wrong query results for the incoming elements. There are two types of false query results: *false positive* or *false negative*. A false positive refers to an element's first arrival being identified as a duplicate, which we should reduce to assist in accurate spread measurement. A false negative happens when a duplicate is falsely reported as a new element. It is the key to achieving short-term memory, and we will discuss it shortly.

We can compute the false positive rate (FPR) with parameter k as the probability that all mapped bits of an element have the value of s. As discussed above, an arbitrary bit in the bit array has the same expectation probability (*i.e.*, 0.5) of being 0 or 1. Thus, the false positive rate can be computed by:

$$FPR = \prod_{j=1}^{k} \Pr\{B[h_j] = s\} = \frac{1}{2^k}. \quad (1)$$

A false negative happens when a duplicate is falsely reported as a new element, *i.e.*, at least one of the k bits correlated with the element is changed to a status different from s. We notice that an element may appear multiple times, but its last appearance will always override the earlier occurrences. Therefore, the false negative rate (FNR) of an element's i-th ($i = 2, 3, \cdots$) arrival is only correlated to the number of distinct elements (denoted by τ) between the (i-1)-th arrival and i-th arrival.

For each bit h correlated with (f, e), we use X and Y to represent the events that a different element changes this bit's status to s or not s and let Z represent the event of no collision, *i.e.*, a different element is not mapped to this bit. The probabilities of those events are $\frac{k}{2m}$, $\frac{k}{2m}$, and $1 - \frac{k}{m}$. Then we can turn the sequence of τ unique elements to an event sequence $\{XYZ\}^{\tau}$, where every item in the sequence can be X, Y, or Z. If we want the status information s at $B[h]$ to be lost after τ unique items, the event sequence must be ended with a Y and a series of Z (length could be 0). The probability of this case, denoted by $FNR_h(\tau)$, can be computed by summering up the probabilities of such event sequences, which is as below:

$$FNR_h(\tau) = \sum_{j=0}^{\tau-1} (\frac{k}{2m})(1 - \frac{k}{m})^j = \frac{1 - (1 - \frac{k}{m})^{\tau}}{2}. \quad (2)$$

Recall that element (f, e) is correlated with k bits, and a false negative is reported if at least one of the k bits is changed to not s. Thus, we can compute the false negative probability $FNR(\tau)$ by one minus the probability that all the k bits preserve to be s as follows:

$$FNR(\tau) = 1 - \prod_{j=1}^{k} (1 - FNR_{h_j}(\tau)). \quad (3)$$

In the above equation, $FNR(\tau)$ is a monotonically increasing function of τ, which means the FNR at an element's arrival will grow with the number of distinct elements that appear after its previous arrival. This property also collaborates with our design goal that STM-DF should filter the duplicates with short-term memory.

C. Duplicate filtration with short-term memory

Before using STM-DF to filter the duplicates from a packet stream, we will first run the stateful initialization to get it prepared for further operations. Then for each packet with flow label f and element label e, STM-DF conducts the duplicate filtration as follows:

First, STM-DF operates a stateful query operation for (f, e) to check whether this element has appeared before. There are two cases to consider: (1) the query operation returns *true*, which means STM-DF has seen this element shortly. At this time, STM-DF will regard this element as a duplicate and take no further actions. (2) the query operation returns a *false* value, which means the element has not appeared before. In this case, STM-DF will regard it as a new element and report it to the sampling module. Meanwhile, STM-DF will insert the new element into its data structure with a stateful insert operation so that this element's subsequent appearances will be recognized as duplicates.

We use an example to explain why this short-term memory property can effectively filter the duplicates of transient elements. Consider a transient element e that appears multiple times during a short period. At its first arrival, STM-DF has a large chance, *i.e.*, $1 - FPR$, to report it as a new element. Then for its subsequent arrivals, STM-DF will misreport each of them as a new element with a probability of $FNR(\tau)$, which is relatively small considering that τ is small. Moreover, when new elements continue to arrive, *i.e.*, τ grows, STM-DF is likely to forget this transient element, achieving the goal of effectively filtering duplicates with a tiny on-chip space.

D. One-memory-access optimization

Considering that simply mapping each element to k bits in the array requires multiple memory accesses, which will result in a performance bottleneck, we introduce a one-memory-access optimization [34] for the proposed STMS. That is, we will divide the bit array into $\frac{m}{w}$ words where each word has w (*e.g.*, 32 or 64) bits and $\frac{m}{w}$ is an integer value. At this time, we will map each element to one word and pseudo-randomly choose k bits in this word, reducing memory access with nearly no influences on duplicate filtration. When implementing STM-DF using FPGA devices, we can create the bit array using SRAM resources, configure the SRAM in dual-port mode (having one read-only port and one write-only port with independent clocks), and set the data width to w bits. In that case, we can ensure that query operation and insert operation will cost exactly one memory access.

V. Algorithm Design

This section presents short-term memory sampling, which can perform accurate per-flow spread measurement with tiny on-chip space and support online queries.

A. Recording

The recording operation is triggered for each arrival packet. The router extracts flow label f and element label e from the packet and passes the element to the on-chip sampling module, which performs the sampling procedure as follows: First, the pre-sampler with a pre-sampling probability p_1 will compute a hash $i = H'(f \oplus e)$ for the incoming element. H' is a hash function with output range $[0, X)$. If $i \leq p_1 X$, this element would be chosen by the pre-sampler. The second phase in the pipeline is STM-DF that accepts elements from the pre-sampler. For each element, STM-DF will check if it has seen this element before and report it as a new element if the query result is false (as described in Section IV). The new element (f, e) reported by STM-DF will be further sent to the off-chip recording module to update its flow record c_f.

We maintain a hash set for each flow f and associate each record c_f with a cardinality field $|c_f|$ at the off-chip recording module to assist effective spread estimation. When an element (f, e) is downloaded, the recorder will check if this flow f has already been assigned with a record and assign one if not. After that, the recorder will update the flow record by inserting the element e into the set, *i.e.*, $c_f \leftarrow c_f \cup \{e\}$, and updating the cardinality field if necessary. Thus, the recording operation for a packet has $O(1)$ runtime.

With this recording operation, we can ensure that each element has the same overall sampling probability p_e of being downloaded to the off-chip recording module at least once. Suppose an element e occurs for v_e times and there are $\tau_{e,2}, \tau_{e,3}, \cdots, \tau_{e,v_e}$ different elements between two adjacent appearances. Notice that the probability of an element being pre-sampled is p_1. If e is pre-sampled and passed to the STM-DF, the numbers of different elements between its two adjacent appearances will be $p_1\tau_{e,2}$, $p_1\tau_{e,3}$, $\cdots$, $p_1\tau_{e,v_e}$. Thus, $p_{2|e}$, the probability that an element e chosen by pre-sampler is downloaded at least once can be computed by one minus the probability of being miss-sampled for v_e times. Considering that miss-sampling probability is FPR for the first arrival and $1 - FNR(p_1\tau_{e,i})$ for the rest of appearances, we have:

$$p_{2|e} = 1 - FPR \times \prod_{i=2}^{v_e}(1 - FNR(p_1\tau_{e,i})). \tag{4}$$

In practice, it is intractable to obtain v_e, $\tau_{e,i}$ for every element with limited memory resource. However, we can replace them with the median values $\widetilde{v}$ and $\widetilde{\tau}$ obtained from the historical traffic data. Let $\mathrm{P}(\widetilde{\tau}_e, \widetilde{v}_e)$ represent the joint distribution of an element's $\widetilde{\tau}_e$ and $\widetilde{v}_e$. We can approximate p_2 for all unique elements as follows:

$$p_2 \approx \sum_{\widetilde{\tau}_e, \widetilde{v}_e} \mathrm{P}(\widetilde{\tau}_e, \widetilde{v}_e) \times (1 - FPR \times (1 - FNR(p_1\widetilde{\tau}_e))^{\widetilde{v}_e - 1}). \tag{5}$$

Combining the above analysis, we can approximate the overall sampling probability p_e using $p_1 \times p_2$, which indicates the probability that an element is chosen by pre-sampler (p_1) and reported by STM-DF at least once (p_2).

We notice that this recording process faces an extreme case where the elements appear periodically (*e.g.*, an element appears once for every 10 seconds). To solve this problem, we can set k of STM-DF to 5 when performing a 5-minute measurement task, which ensures a small variance in different elements' actual sampling probabilities. At this time, p_e is a reliable approximate of the sampling probabilities for all elements, hence we can provide a high estimation accuracy for all flows.

B. Estimation

When receiving a query for the spread of flow f, we first retrieve its record c_f from the off-chip recorder. If no record is found for f, we regard this flow as an empty flow and return an estimation of 0. Otherwise, we can estimate flow f's spread by dividing the cardinality of c_f with the overall sampling probability p_e. Formally:

$$\widehat{n_f} = \frac{|c_f|}{p_e} = \frac{|c_f|}{p_1 \times p_2}. \tag{6}$$

Besides, the time complexity of spread estimation is also $O(1)$ since retrieving the cardinality field $|c_f|$ only takes one memory access. This ensures that STMS is sufficient to answer any online per-flow spread queries.

VI. Performance Analysis

A. On-chip memory usage and communication cost

When performing per-flow spread measurement, STMS maintains a bit array in the on-chip memory and continuously sends flow-element tuples (f, e) to the off-chip recording module. We quantify the on-chip memory usage using the bit array size m of STM-DF. Besides, we quantify the communication cost using the offload speed r (/Mpps), which refers to the number of elements we offload to the off-chip recording part in one second. We can compute r by multiplying the packet arriving rate with the packet-level sampling probability p_s.

We can estimate the offload speed based on system parameters p_1, m, and k as follows: For an element e that appears v_e times, it has a probability of p_1 to be chosen by pre-sampler. The probabilities of this element been offloaded to the off-chip recording module at first appearance and subsequent occurrences are $p_1 \times (1 - FPR)$ and $p_1 \times FNR(p_1\tau_{e,i})$, $i = 2, 3, \cdots, v_e$. Thus, the average downloading times for this element will be $p_1 \times (1 - FPR + \sum_{i=2}^{v_e} FNR(p_1\tau_{e,i}))$. Again, we can approximate the average sampling times of all elements using median values $\widetilde{v_e}$ and $\widetilde{\tau_e}$ and joint distribution $\mathrm{P}(\widetilde{\tau}_e, \widetilde{v}_e)$. Then we can compute the packet-level sampling probability p_s for all packets as follows:

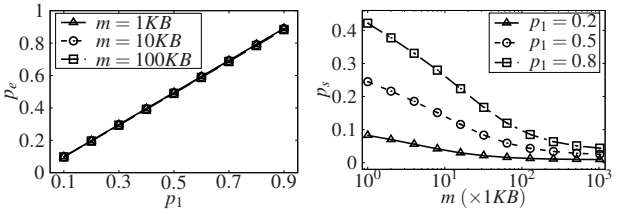
$$p_s \approx \sum_{\widetilde{\tau}_e, \widetilde{v}_e} \mathrm{P}(\widetilde{\tau}_e, \widetilde{v}_e) \frac{p_1 \times (1 - FPR + (\widetilde{v}_e - 1)FNR(p_1\widetilde{\tau}_e))}{\widetilde{v}_e}. \tag{7}$$

Supposing the packet arrival rate is t Mpps, we can estimate the offload speed r during the measurement by $r = t \times p_s$. Usually, we need to determine the system parameters so that the estimation accuracy is guaranteed while the communication cost does not overwhelm the transfer link, *i.e.*, offload speed r is no larger than the desired threshold r_{th}. Also, we find out

the estimation accuracy is controlled by the overall sampling probability, which will be discussed detailedly in the following parts. Anyway, a larger overall sampling probability will always promise better performance. Thus, system parameter selection can be transformed to minimize the on-chip memory usage m while ensuring the overall sampling probability and communication cost satisfy the requirements:

$$\min m \quad \text{s.t.} \begin{cases} p_1 \times p_2 \geq p_{min} \\ t \times p_s \leq r_{th}, \end{cases} \tag{8}$$

where the first line bounds that the overall sampling probability of a distinct element, *i.e.*, $p_e = p_1 \times p_2$, is no less than the desired probability p_{min} raised by performance requirements. The second line constraints that the offload speed is no more than threshold r_{th}.



(a) p_e under different parameters (b) p_s under different parameters

Fig. 3: Curve of p_e and p_s under different p_1 and m.

In Fig. 3, we plot the curves of p_e and p_s under different system parameters. The joint distribution of $\mathrm{P}(\widetilde{\tau}_e, \widetilde{v}_e)$ is learned from the one-minute trace download from CAIDA [30], and we set k to 5 in our implementations. From the left subplot, we find out that p_e is almost irrelevant to m, and is increased when p_1 increases. The right plot denotes the relationship between p_s and system parameters. When p_1 is fixed, p_s is a decreasing function of m. Combining the above analysis, we can solve the parameter selection problem above by first selecting the minimal value of p_1 that makes $p_e \geq p_{min}$. After that, we can choose the minimal value of m that satisfies the second constraint.

B. Relative error bound

Now we are going to explain how to optimize the system parameters (m, p_1, k) to bound the relative errors of spread estimations. Considering an arbitrary flow f with the spread of n_f larger than a pre-defined threshold T; our goal is to bound its relative error by δ with a probability of at least $1-\epsilon$. Formally, the estimated spread $\widehat{n_f}$ should satisfy:

$$\Pr\{|1 - \frac{\widehat{n_f}}{n_f}| \leq \delta\} \geq 1 - \epsilon. \tag{9}$$

We model the recording procedure as a sequence of Bernoulli trials, where each distinct element represents a trial. At this time, we regard the event that an element is recorded off-chip as a successful trial. Thus, $|c_f|$, the number of recorded distinct elements, follows a Bernoulli distribution $\mathrm{Bernoulli}(n_f, p_e)$ parameterized by $p_e = p_1 \times p_2$.

If we want to bound the relative error of estimated spread by δ, we should ensure that the number of successes, *i.e.*, $|c_f|$, should be within $\lceil (1-\delta) n_f p_e \rceil$ and $\lfloor (1+\delta) n_f p_e \rfloor$. Thus, we can estimate the probability that a flow's estimated spread locates in $[(1-\delta)n_f, (1+\delta)n_f]$ with the following equation:

$$\Pr\{|1-\frac{\widehat{n_f}}{n_f}| \leq \delta\} = \sum_{i=\lceil (1-\delta) n_f p_e \rceil}^{\lfloor (1+\delta) n_f p_e \rfloor} \binom{n_f}{i} p_e^i (1-p_e)^{n_f - i}. \tag{10}$$

Apparently, the bound probability is an increasing function of n_f, which means if we can ensure that the relative error bound works for flows with spread T, we can guarantee the same error bound for all flows with spread larger than T. Let $p_T(\delta, p_e)$ represent the value of $\Pr\left\{|1 - \frac{\widehat{n_f}}{n_f}| \leq \delta\right\}$ when $n_f = T$. We can determine p_{min}, the minimum value for p_e, as the smallest value that makes $p_T(\delta, p_e)$ no less than $1-\epsilon$:

$$p_{min} = \min\{p_e \,|\, p_T(\delta, p_e) \geq 1 - \epsilon; 0 < p_e < 1\}. \tag{11}$$

After that, we can optimize the system parameters m, p_1, k to bound the relative errors as we discussed in Section VI-A.

VII. Experimental Results

A. Experimental Setup

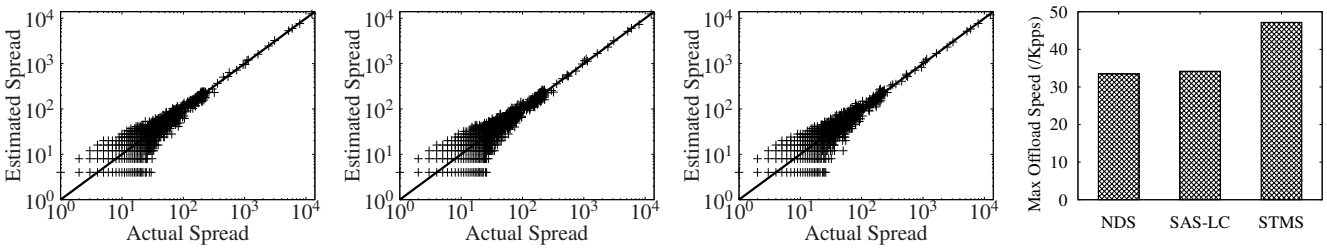
In the experiments, we evaluate the performance of the proposed solution STMS using two different datasets, respectively, the 1-minute and 5-minute Internet traces downloaded from CAIDA [30]. The 1-minute dataset contains 589740 per-source flows and 907463 elements (destination addresses). The 5-minute dataset contains 1689780 per-source flows and 3150740 elements. In the experiments, we use NDS [27] and SAS-LC [28] as baselines and compare them with STMS. We have released the source code on GitHub [35].

We implement the proposed short-term memory sampling on a NetFPGA-embedded prototype, where a NetFPGA-1G-CML development board (with 326K logic units, 16 Mbits Block RAM, and a clock rate of 100MHz) connects with the workstation (with Ryzen7 1700 @3.0GHz CPU and 64GB RAM) through PCIe Gen2 X4 lanes. In our implementation, the throughput threshold of the on-chip sampling module and the off-chip recording module are separately 100Mpps and 14.95Mpps. To keep up with the line speed, the element offloading speed should not exceed the throughput threshold of off-chip recording, *i.e.*, the packet-level sampling probability p_s multiplies 100Mpps should not be larger than 14.95Mpps. Thus, we have $p_s \leq 0.1495$.

Further, we evaluate the response time of STMS to answer an arbitrary per-flow spread query. We execute the off-chip spread estimations on a machine with two six-core Intel Xeon E5-2643v4 @3.40GHz CPU and 256GB RAM. The experimental results show that the response time of STMS for an arbitrary flow is less than 1μs, which is efficient enough for any online query.

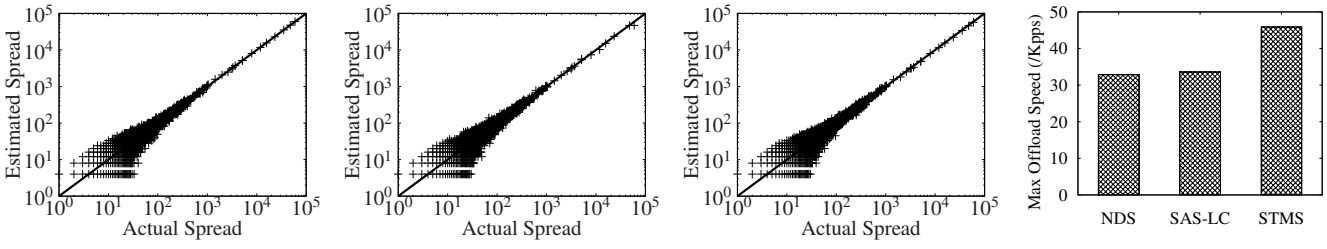
B. Memory requirements

We compare STMS, NDS, and SAS-LC in terms of the on-chip memory requirements when setting different relative error bounds (as illustrated in Section VI-B). The required memory of NDS and SAS-LC is computed according to the parameter



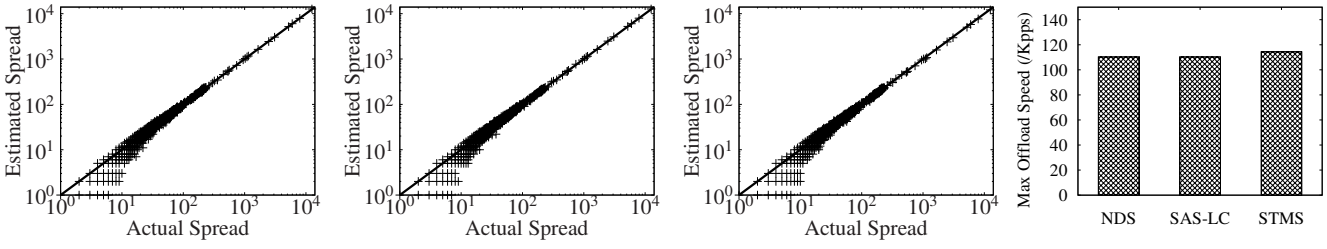
(a) NDS, $m = 0.076$MB (b) SAS-LC, $m = 0.075$MB (c) STMS, $m = 0.001$MB (d) Maximum offload speed

Fig. 4: Spread estimation accuracy of NDS, SAS-LC, and STMS on 1-minute dataset when $\delta = 0.2, \epsilon = 0.1, T = 200$



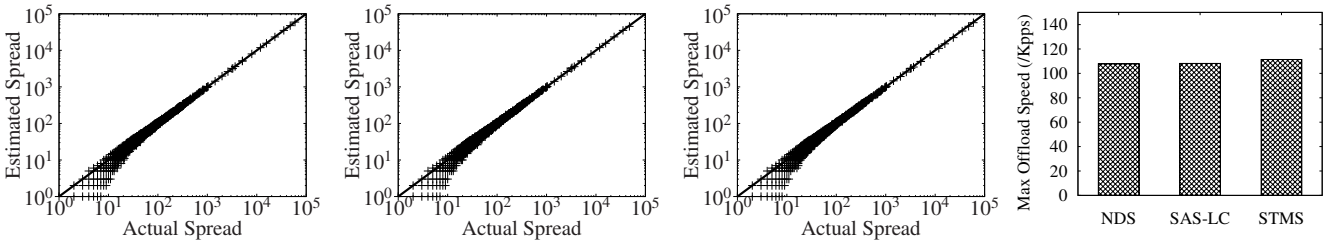
(a) NDS, $m = 0.264$MB (b) SAS-LC, $m = 0.253$MB (c) STMS, $m = 0.001$MB (d) Maximum offload speed

Fig. 5: Spread estimation accuracy of NDS, SAS-LC, and STMS on 5-minute dataset when $\delta = 0.2, \epsilon = 0.1, T = 200$



(a) NDS, $m = 0.449$MB (b) SAS-LC, $m = 0.449$MB (c) STMS, $m = 0.041$MB (d) Maximum offload speed

Fig. 6: Spread estimation accuracy of NDS, SAS-LC, and STMS on 1-minute dataset when $\delta = 0.1, \epsilon = 0.05, T = 100$



(a) NDS, $m = 1.559$MB (b) SAS-LC, $m = 1.515$MB (c) STMS, $m = 0.041$MB (d) Maximum offload speed

Fig. 7: Spread estimation accuracy of NDS, SAS-LC, and STMS on 5-minute dataset when $\delta = 0.1, \epsilon = 0.05, T = 100$

selection methods proposed in [27] and [28]. The memory usage of STMS is chosen by the method in Section VI, *i.e.*, choosing the minimal memory size m that satisfies the desired error bound with a packet-level sampling probability (p_s) no larger than 0.1495.

TABLE I shows the memory usage of three algorithms on two datasets with respect to δ, ϵ, and T. We can observe that, under all settings, STMS can achieve the desired performance with extremely small on-chip memory usage, outperforming the baselines by saving $90\%\sim99\%$ of the memory usage. Another observation is that the memory usages of NDS and SAS-LC are proportional to the measurement period since there are more elements in a longer period. Unlike these baselines, the memory usage of STMS is irrelevant to the measurement period because it utilizes a novel short-term memory strategy to address the massive transient elements.

C. Estimation Accuracy

Fig. 4 - Fig. 7 and TABLE II - TABLE V compare our method STMS with NDS and SAS-LC on spread estimation accuracy. We set two different relative bounds ($\delta = 0.2$, $\epsilon = 0.1$, $T = 200$, and $\delta = 0.1$, $\epsilon = 0.05$, $T = 100$) on both 1-minute dataset and 5-minute dataset. Their memory usages are set as we described above. The other parameters are computed according to [27], [28], and this work.

Fig. 4 and TABLE II depict the experimental results on 1-minute dataset with constraint ($\delta = 0.2$, $\epsilon = 0.1$, $T = 200$). In this set of experiments, the on-chip memory usages of

TABLE I: On-chip memory requirements (MB) of NDS, SAS-LC, and STMS for 1-minute and 5-minute datasets.

	T	1-minute dataset			5-minute dataset		
		NDS	SAS-LC	STMS	NDS	SAS-LC	STMS
$\delta = 0.2, \epsilon = 0.1$	50	0.181	0.188	**0.013**	0.628	0.629	**0.013**
	100	0.110	0.111	**0.003**	0.382	0.373	**0.003**
	150	0.090	0.090	**0.001**	0.312	0.303	**0.001**
	200	0.076	0.075	**0.001**	0.265	0.253	**0.001**
	250	0.067	0.067	**0.001**	0.233	0.226	**0.001**
	300	0.063	0.062	**0.001**	0.218	0.209	**0.001**
$\delta = 0.1, \epsilon = 0.05$	50	0.824	0.821	**0.057**	2.862	2.768	**0.057**
	100	0.449	0.449	**0.041**	1.560	1.515	**0.041**
	150	0.316	0.315	**0.030**	1.097	1.065	**0.030**
	200	0.251	0.251	**0.023**	0.872	0.866	**0.023**
	250	0.212	0.211	**0.017**	0.735	0.734	**0.017**
	300	0.184	0.183	**0.013**	0.638	0.624	**0.013**

TABLE II: Actual relative error bound for spread estimation on 1-minute dataset ($\delta = 0.2, \epsilon = 0.1, T = 200$)

spread / algorithm	all flows	$1 \sim 200$	$200 \sim 1000$	$1000 \sim 5000$	$5000 \sim 10000$
NDS	3.119	3.119	0.198	0.088	0.035
SAS-LC	3.100	3.100	0.195	0.158	0.077
STMS	3.119	3.119	0.187	0.091	0.033

TABLE III: Actual relative error bound for spread estimation on 5-minute dataset ($\delta = 0.2, \epsilon = 0.1, T = 200$)

spread / algorithm	all flows	$1 \sim 200$	$200 \sim 1000$	$1000 \sim 10000$	$10000 \sim 100000$
NDS	3.119	3.119	0.170	0.095	0.015
SAS-LC	3.125	3.125	0.184	0.076	0.068
STMS	3.119	3.119	0.195	0.076	0.041

TABLE IV: Actual relative error bound for spread estimation on 1-minute dataset ($\delta = 0.1, \epsilon = 0.05, T = 100$)

spread / algorithm	all flows	$1 \sim 100$	$100 \sim 1000$	$1000 \sim 5000$	$5000 \sim 10000$
NDS	1.000	1.000	0.076	0.018	0.008
SAS-LC	1.000	1.000	0.063	0.018	0.008
STMS	1.000	1.000	0.091	0.076	0.035

TABLE V: Actual relative error bound for spread estimation on 5-minute dataset ($\delta = 0.1, \epsilon = 0.05, T = 100$)

spread / algorithm	all flows	$1 \sim 200$	$200 \sim 1000$	$1000 \sim 10000$	$10000 \sim 100000$
NDS	1.000	1.000	0.058	0.022	0.004
SAS-LC	1.000	1.000	0.066	0.019	0.005
STMS	1.000	1.000	0.070	0.039	0.044

NDS and SAS-LC are 0.076MB and 0.075MB, while our method STMS can achieve the desired estimation accuracy with only 0.001MB on-chip space and reduce around 99% of two baselines' memory usage. We show the estimation results in three plots (Fig. 4(a) - Fig. 4(d)), where each point represents a flow. The x-axis denotes the actual spread, and the y-axis denotes the estimated spread. Thus, the estimation result is more accurate when the point is closer to the equality line $y = x$. From these plots, we find out that the estimation accuracy of the three algorithms grows when the flow spread increases. Then in TABLE II, we present the actual relative error bound for three algorithms, which refers to a particular threshold that ensures precisely $1 - \epsilon$ of the flows have a relative error no more than the threshold. From the table, we find out that the actual relative error bounds for flows with a spread larger than T (=200) are all smaller than the desired value δ (=0.2), which means all three algorithms have satisfied the desired accuracy constraints. Besides, in Fig. 4(d), we present the maximum offload speed of three algorithms during the measurement period, which is computed based on the number of downloaded elements in a time slot. We find out that STMS has a maximum offload speed (47.18Kpps) slightly higher than the NDS (33.47Kpps) and SAS-LC (34.19Kpps). However, this will not result in a performance bottleneck since the parameter selection method (proposed in Section VI-A) can prevent the offloaded elements from overwhelming the off-chip recording module. More specifically, the packet-level sampling probability p_s of STMS is 0.0798 in this set of experiments, which is smaller than the threshold (0.1495).

The second set of experiments are performed on 5-minute data set with a constraint of $\delta = 0.2$, $\epsilon = 0.1$, $T = 200$. The experimental results are presented in Fig. 5 and TABLE III. At this time, the memory usage of NDS, SAS-LC, and STMS are 0.264MB, 0.253MB, and 0.001MB. Compared with the first set of experiments, we find that NDS and SAS-LC demand larger on-chip memory with a longer measurement period. Unlike the baselines, STMS can satisfy the same accuracy constraint with only 0.001MB memory usage, which is identical to its memory usage in the first set of experiments. This demonstrates that STMS achieves our design goal of using small and constant on-chip memory to perform accurate per-flow spread estimation.

Fig. 6 - Fig. 7 and TABLE IV - TABLE V depict the experimental results under constraint ($\delta = 0.1$, $\epsilon = 0.05$, $T = 100$), from which we can obtain similar observations as in the above experiments. Clearly, our solution (STMS) is the winner. Compared to the state-of-the-art, it reduces up to 99% of the on-chip memory usage when providing the same performance assurances on the spread-estimation accuracy.

VIII. Conclusion

This paper proposes an efficient short-term memory sampling method for per-flow spread measurement, providing accurate estimation results for online spread queries with minimal on-chip memory usage. The experimental results based on real Internet traffic traces demonstrate that our new estimator can be flexibly configured to meet the measurement interests of different applications and can work efficiently in an extremely small on-chip memory space (such as 1KB), while the best existing work will fail.

Acknowledgments

The corresponding authors of this paper are He Huang and Yu-E Sun. This work was supported by National Natural Science Foundation of China under Grant No. 62072322, No. 61873177, No. U20A20182, and No. 62102275 , Natural Science Foundation of Jiangsu Province under Grant No. BK20210706 and No. BK20210704, and Jiangsu Planned Projects for Postdoctoral Research Funds under Grant No. 2021K165B.

REFERENCES

[1] H. Xu, Z. Yu, C. Qian, X. Li, Z. Liu, and L. Huang, "Minimizing flow statistics collection cost using wildcard-based requests in SDNs," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3587–3601, 2017.

[2] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2020)*, 2020, pp. 1339–1348.

[3] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond sketch: Accurate per-flow measurement for big streaming data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2650–2662, 2019.

[4] R. Ben Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k, frequency, and volume estimation," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1432–1445, 2019.

[5] C. Hu, B. Liu, S. Wang, J. Tian, Y. Cheng, and Y. Chen, "ANLS: adaptive non-linear sampling method for accurate flow size measurement," *IEEE Transactions on Communications*, vol. 60, no. 3, pp. 789–798, 2012.

[6] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, Y. Cheng, and H. Wu, "Discount counting for fast flow statistics on flow size and flow volume," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 970–981, 2014.

[7] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile, "Generalized sketch families for network traffic measurement," *Proc. of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–34, 2019.

[8] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. of IEEE International Conference on Network Protocols (ICNP 2015)*, 2015, pp. 1–10.

[9] R. Cohen and Y. Nezri, "Cardinality estimation in a virtualized network device using online machine learning," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 2098–2110, 2019.

[10] L. Tang, Q. Huang, and P. P. C. Lee, "Spreadsketch: Toward invertible and network-wide detection of superspreaders," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2020)*, 2020, pp. 1608–1617.

[11] T. Li, S. Chen, W. Luo, and M. Zhang, "Scan detection in high-speed networks based on optimal dynamic bit sharing," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2011)*, 2011, pp. 3200–3208.

[12] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Du, H. Wang, and Q. Xiao, "Spread estimation with non-duplicate sampling in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2073–2086, 2021.

[13] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Zhou, W. Yang, S. Tang, H. Xu, and Y. Qiao, "An efficient k-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1463–1476, 2020.

[14] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[15] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. of the 16th International Conference on Extending Database Technology (EDBT 2013)*, 2013, pp. 683–692.

[16] P. Lieven and B. Scheuermann, "High-speed per-flow traffic measurement with probabilistic multiplicity counting," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2010)*, 2010, pp. 1–9.

[17] M. Yoon, T. Li, S. Chen, and J. kwon Peir, "Fit a spread estimator in small memory," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2009)*, 2009, pp. 504–512.

[18] Y. Li, H. Wu, T. Pan, H. Dai, J. Lu, and B. Liu, "CASE: cache-assisted stretchable estimator for high speed per-flow measurement," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2016)*, 2016, pp. 1–9.

[19] R. Jang, S. Moon, Y. Noh, A. Mohaisen, and D. Nyang, "Instameasure: Instant per-flow detection using large in-dram working set of active flows," in *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS 2019)*, 2019, pp. 2047–2056.

[20] T. Yang, J. Xu, X. Liu, P. Liu, L. Wang, J. Bi, and X. Li, "A generic technique for sketches to adapt to different counting ranges," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2019)*, 2019, pp. 2017–2025.

[21] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Scream: Sketch resource allocation for software-defined measurement," in *Proc. of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT 2015)*. Association for Computing Machinery, 2015, pp. 1–13.

[22] H. Huang, Y. Sun, S. Chen, S. Tang, K. Han, J. Yuan, and W. Yang, "You can drop but you can't hide:k-persistent spread estimation in high-speed networks," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2018)*, 2018, pp. 1889–1897.

[23] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a compact spread estimator in small high-speed memory," *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 5, pp. 1253–1264, 2011.

[24] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, "Highly compact virtual active counters for per-flow traffic measurement," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2018)*, 2018, pp. 1–9.

[25] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir, "Randomized error removal for online spread estimation in data streaming," *Proc. VLDB Endow.*, vol. 14, no. 6, p. 1040–1052, 2021.

[26] H. Dai, M. Shahzad, A. X. Liu, M. Li, Y. Zhong, and G. Chen, "Identifying and estimating persistent items in data streams," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2429–2442, 2018.

[27] Y. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao, "Online spread estimation with non-duplicate sampling," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2020)*, 2020, pp. 2440–2448.

[28] Y. Du, H. Huang, Y. Sun, S. Chen, and G. Gao, "Self-adaptive sampling for network traffic measurement," in *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2021)*, 2021, pp. 1–10.

[29] Y. Du, H. Huang, Y.-E. Sun, A. Liu, G. Gao, and B. Zhang, "Online high-cardinality flow detection over big network data stream," in *Proc. of the International Conference on Database Systems for Advanced Applications (DASFAA 2021)*, 2021, pp. 405–421.

[30] CAIDA, "The CAIDA UCSD Anonymized Internet Traces 2016," http://www.caida.org/data/passive/passive_2016_dataset.xml, accessed July 28, 2019.

[31] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

[32] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*, 2006, p. 25–36.

[33] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams," *Proc. VLDB Endow.*, vol. 6, no. 8, p. 589–600, 2013.

[34] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. of the International Conference on Management of Data (SIGMOD 2018)*, 2018, pp. 741–756.

[35] "Source code related to STMS," https://github.com/duyang92/STMS, 2021.