SketchFlow: Per-Flow Systematic Sampling Using Sketch Saturation Event

Rhongho Jang
University of Central Florida (UCF) & Inha University
r.h.jang@knights.ucf.edu

DaeHong Min
Inha University
Incheon, Korea
mang@isrl.kr

SeongKwang Moon
Inha University
Incheon, Korea
skmoon@isrl.kr

David Mohaisen
University of Central Florida
Orlando, USA
mohaisen@ucf.edu

DaeHun Nyang
Inha University
Incheon, Korea
nyang@inha.ac.kr

***Abstract*—Sampling is a powerful tool to reduce the processing overhead in various systems. NetFlow uses a local table for counting records per flow, and sFlow sends out the collected packet headers periodically to a collecting server over the network. Any measurement system falls into either one of these two models. To reduce the overhead, as in sFlow, simple random sampling (SRS) has been widely used in practice because of its simplicity. However, SRS provides non-uniform sampling rates for different fine-grained flows (defined by 5-tuple), because it samples packets over an aggregated data flow (defined by switch port or VLAN). Consequently, some flows are sampled more than the designated sampling rate (resulting in over-estimation), and others are sampled fewer (resulting in under-estimation). Starting with a simple idea that "independent per-flow packet sampling provides the most accurate estimation of each flow", we introduce a new concept of per-flow systematic sampling, aiming to provide the same sampling rate across all flows. In addition, we provide a concrete sampling method called SketchFlow, which approximates the idea of the per-flow systematic sampling using a sketch saturation event. We demonstrate SketchFlow's performance in terms of accuracy, sampling rate, and overhead using real-world datasets, including a backbone network trace, I/O trace, and Twitter dataset. Experimental results show that SketchFlow outperforms SRS (*i.e.,* sFlow) and the non-linear sampling method while requiring a small CPU overhead to measure high-speed traffic in real-time.**

I. Introduction

The simple random sampling (SRS) has played an important role in network traffic measurement, resulting in standards such as Sampled NetFlow [28] and sFlow [32]. For instance, the Sampled NetFlow samples packets to reduce the CPU overhead of switches to prevent delay in routing decisions. sFlow uses simple random sampling to reduce meta-data transmission over the network. Sampling has been comprehensively studied, since the work of Claffy *et al.* [4], which uses sampling for gathering network statistics. As an alternative solution, however, sketches have been introduced by Morris [26] and Flajolet *et al.* [10]. Since then, many works have been conducted to enhance sketches' accuracy while reducing their overhead [9], [17], [29], [37]. A comparative study of sampling and sketches has been done by Tune *et al.* [36].

Sampling is a practical solution in many areas, such as network measurement and high-volume data analysis (categories of sampling are shown in Fig. 1). As such, it has played a significant role as a filter to reduce the burden on the flow record table (e.g., in NetFlow) and to lessen the network

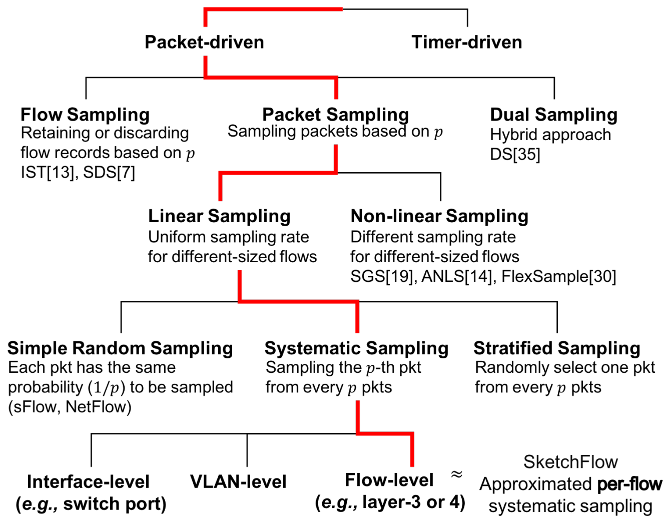

Packet-driven
Timer-driven
Flow Sampling
Retaining or discarding flow records based on p
IST[13], SDS[7]
Packet Sampling
Sampling packets based on p
Dual Sampling
Hybrid approach
DS[35]
Linear Sampling
Uniform sampling rate for different-sized flows
Non-linear Sampling
Different sampling rate for different-sized flows
SGS[19], ANLS[14], FlexSample[30]
Simple Random Sampling
Each pkt has the same probability ($1/p$) to be sampled
(sFlow, NetFlow)
Systematic Sampling
Sampling the p-th pkt from every p pkts
Stratified Sampling
Randomly select one pkt from every p pkts
Interface-level
(*e.g.,* switch port)
VLAN-level
Flow-level
(*e.g.,* layer-3 or 4)
≈
SketchFlow
Approximated **per-flow** systematic sampling

Fig. 1. Design space of SketchFlow.

bandwidth overhead (e.g., in sFlow). Therefore, maintaining a stable task reduction rate is a crucial part of evaluating sampling algorithms, where the reduction of the influx of elements is determined by the *sampling rate*, which also leads to the well-known trade-off between accuracy and overhead. A large sampling rate (e.g., $1/10$) achieves high accuracy by conducting fine-grained sampling by obtaining samples more frequently. On the contrary, a small sampling rate (e.g., $1/10,000$) provides coarse-grained samples (*i.e.,* relatively low accuracy), but fewer samples are taken. To provide a better trade-off, many sampling strategies have been proposed. Claffy *et al.* [4] showed that timer-driven sampling does not perform as well as event-driven (or packet-driven) sampling. Among packet-driven sampling methods, most research works are on packet sampling, but flow thinning or flow sampling has been shown to be better in terms of its accuracy [13]. However, it heavily relies on additional information, such as TCP SYN/SEQ signals. That means the sampling is not general enough to be used for other purposes such as UDP traffic measurement–QUIC (Quick UDP Internet Connections) has occupied 7% of the global traffic in 2016 (and more than 7.8% as of late 2018) [31]. Moreover, such an approach has to manage flow labels in a hash table, which is another challenge.

Packet sampling is categorized into linear and non-linear sampling, per Fig. 1. The linear sampling is featured by uniformly sampling $1/p$ packets of a data stream, where p is the sampling interval and $1/p$ is the sampling rate. According to Claffy *et al.* [4], the simple random sampling, stratified sampling, and systematic sampling can be applied as sampling strategies. Recent works have focused on how to apply a non-linear sampling rate according to the flow size [14], [19], [30], where mouse flows get sampled more often and elephant flows less often using a non-linear function based on the flow size. On the downside, the non-linearity in the sampling rate substantially increases the overhead by sampling small flows heavily to guarantee the accuracy for a traffic distribution.

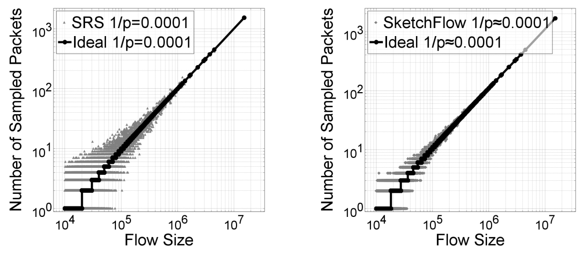
Sketches are compact data structures that use probabilistic counters for approximate estimation of spectral densities of flows [5], [18], [21]–[23], [29]. Sketch-based algorithms have been shown more accurate in estimation than sampling approaches while using a small amount of memory. The higher accuracy of sketches is owing to its per-flow nature of estimation. However, research on sketch-based estimation has mainly focused on the sketch itself: the very nature of sketch to use only a small amount of memory prohibits it from being used for processing large scale data. More specifically, once a sketch is saturated, it cannot count at all. Consequently, sketch-based measurement algorithms have been used in a limited way such as for anomaly detection (e.g., heavy hitter, super spreader, *etc.*) within a short time frame [15], [22], [24], [39]. Also, decoding a sketch is computational heavy, thus cannot be done in data-plane, and thus sketches usually are delivered to a server with enough computing power for decoding, which inevitably introduces a control loop delay.

Our goal is to design a new sketch-based sampling algorithm, called SketchFlow, to provide a better trade-off between accuracy and overhead for a given sampling rate of $1/p$. SketchFlow performs an approximated systematic sampling for fine-grained flows (e.g., layer-4 flows) independently. As a result, almost exactly $1/p$ packets from each and every flow will be sampled. This property is in contrast to SRS, in which the sampling rate across different flows in a data stream is not guaranteed. SketchFlow provides a high estimation accuracy, processes high-speed data in real-time, and is general enough to be used for many estimation purposes without any application-specific information. The core idea of SketchFlow is to recognize a sketch saturation event for a flow and sample only the triggering packets. The saturated sketch for the flow is reset so that it can be reused. Therefore, SketchFlow can be seen as a sampler as well as a sketch. SketchFlow, however, does not work alone as a sketch measuring the whole data stream, but as a general sampler to NetFlow and sFlow.

In summary, our contributions in this paper are as follows: (i) We introduce the new notion of per-flow systematic packet sampling for a precise sampling. See Fig. 1 for how our contribution fits within the literature. (ii) We propose a new framework using the per-flow sketch saturation event as a sampling signal of the flow, whereby only a signaling packet is sampled from the flow, and the saturated sketch is emptied for the next round sampling. This use of a sketch as a sampler is new in the sense that a per-flow sketch now works as a per-flow systematic sampler, and the sketch saturation is not any more an issue. We note, however, that a sketch is an approximate per-flow counter thus a sampling algorithm under the framework is only an approximate per-flow systematic sampler. A new instance can be designed using any better sketch when available. (iii) We realize an approximate version of per-flow systematic packet sampling called SketchFlow. For this purpose, a new per-flow sketch algorithm is presented, which can encode and decode flows in real-time. Multi-layer sketch design is applied for scalable sampling. (iv) We demonstrate SketchFlow's performance in terms of the stable sampling rate, accuracy, and overhead using real-world datasets, including a backbone network trace, hard disk I/O trace, and Twitter dataset.

II. Motivation: Flow-aware Sampling vs. Flow-oblivious Sampling

The bottleneck of NetFlow is the processing capacity for the local table, and that of sFlow is the network capacity. To address the bottleneck, the widely-adopted simple random sampling (SRS) is used with a very small overhead. In theory, SRS guarantees each packet has an equal chance to be sampled. However, the general usage of SRS is for sampling over the interface or VLAN, which collects coarse samples without considering the individual fine-grained flows, such as a flow defined by the 5-tuple. Consequently, some flows are sampled more than the designated sampling rate, resulting in over-estimation, while others suffer from under-estimation. We note that, although the main purpose of traffic measurement is mostly to obtain per-flow statistics such as the spectral density of flow size and distribution, sampling has been applied to data streams aggregating all the flows, rather than individual flows. SRS samples packets with $1/p$ over the entire data stream, although it cannot guarantee the sampling rate to be $1/p$ for each flow. For per-flow statistics, however, the estimation accuracy is ideal when exactly f/p packets for each flow are sampled (See the solid lines in Fig. 2), where f is the flow size and $1/p$ is the sampling rate. If more or fewer packets than f/p are sampled for a flow, it leads to over- or under-estimation of the actual flow size, because the number of the sampled packets is multiplied by p to estimate f. Therefore, the best strategy is to keep the per-flow sampling rate identical across flows. To that end, we propose the *per-flow systematic packet sampling*, which is a method to sample every p-th packet within a flow, whereas the well-known packet-level systematic sampling is to sample every p-th packet over the entire data stream. Fig. 2(a) shows the number of sampled packets according to flow size for a given sampling rate. The sampling quality is captured by how close the grey dot (the number of actually-sampled packets) is from the solid line (the number of ideally-sampled packets) in this figure. Here, we see that the sampling quality of the flow-oblivious sampling, such as the simple random sampling (*i.e.,* SRS), is much poorer than that of the per-flow systematic sampling (*i.e., ideal*), which is a flow-aware sampling algorithm ((i)).



(a) Flow-oblivious sampling (b) Flow-aware sampling

Fig. 2. Number of sampled packets compared to exact per-flow systematic sampling (*i.e.,* ideal): the estimation of SketchFlow is more accurate than the simple random sampling (SRS).

The complexity of the per-flow systematic sampling problem is equivalent to the per-flow counting problem, which means we still have to pay a large amount of memory/computations for the flow table (*i.e.,* fail to reduce the complexity). To address this issue, we propose a sketch saturation-driven per-flow systematic sampling framework. Our framework utilizes a sketch to reduce the complexity of the per-flow counting problem. The sketch in the framework, however, is used to estimate a sampling interval of a flow, rather than the flow size. Therefore, the sketch does not need to be large to hold the whole flows' total length, but it would be sufficient even when small because it holds only concurrent flows' sampling intervals and resets. When a packet arrives, the sketch encoding algorithm recognizes its flow to sketch individual flows on a small memory in real-time. When the sketch space is saturated, the triggering packet is sampled to a flow table (e.g., NetFlow) or to a collector (e.g., sFlow), and the saturated sketch is emptied for the next round sampling. One can build a per-flow systematic packet sampling algorithm easily from the generic framework by defining an online-encodable/decodable sketch algorithm. Since a sketch for per-flow estimation of the sampling interval has an approximate counting structure, a sampling algorithm from the framework is an approximate version of the per-flow systematic sampling, providing a very high accuracy in per-flow statistics while reducing the overhead (both tables and network bandwidth) by keeping the sampling rate consistent across flows (ii).

SketchFlow is a concrete example of the framework. Fig. 2(b) illustrates the accuracy of SketchFlow. For each flow, the fraction of the sampled packet number over the flow size is almost equivalent to the sampling rate of $1/p$. Moreover, the variance of SketchFlow is much smaller than flow-oblivious sampling schemes (Fig. 2(a)). In addition, SketchFlow can provide mouse flow samples by stacking these flows to trigger sampling events (See section V-D). To sum up, legacy SRS can be replaced by SketchFlow in many applications such as network monitoring (e.g., NetFlow and sFlow), big data analytics (e.g., PowerDrill [12]), and social network service data analysis (e.g., Twitter and Facebook) for better performance.

III. Sketch-based Per-flow Systematic Sampling

We present SketchFlow, an instance of our framework using per-flow sketch to trigger per-flow sampling. SketchFlow is an approximate per-flow systematic sampling (iii).

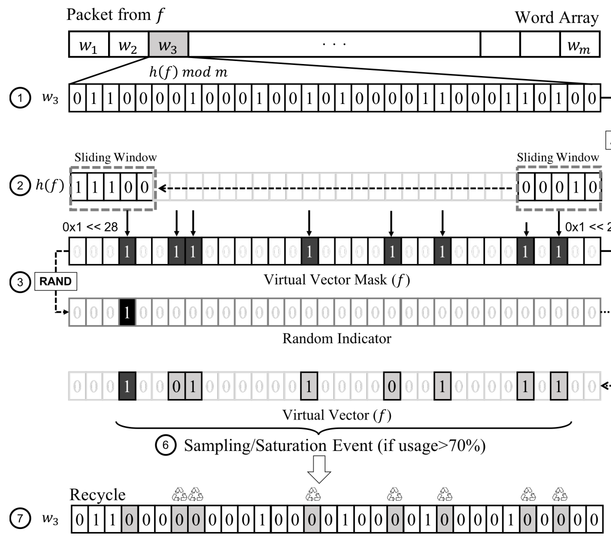


Fig. 3. The overview of SketchFlow

A. Encoding: Data Structure and Overview

Fig. 3 shows an overview of SketchFlow designed to perform an approximate per-flow sampling using a small amount of memory. We constructed SketchFlow using a word array, which is initialized to all 0's. When a flow f arrives, a word from the word array will be selected using the 5-tuple hash value $\mathtt{h}(f)$ (①) , and then s bits of the register (*i.e.,* vector mask) are allocated to f according to the partial output (sliding window) of $\mathtt{h}(f)$ (②). The virtual vector is extracted by doing "Bitwise AND" between register and vector mask (④). For each packet from f, a randomization technique [29] is used for multiplicity counting. That is, one bit position of the vector is randomly flipped to 1 by ③-⑤. A sampling event of each flow is triggered when the usage of the vector exceeds its limit (*i.e.,* vector saturation) (⑥), and then the estimated value of the average number of packets to saturate the vector becomes the sampling interval $\hat{p}$. In SketchFlow, Linear counting (LC) is used for volume estimation by $\hat{p} = -m\ln(V)$, where m is the number of bits (or memory size), and V is the fraction of 0's remaining in the vector. Our approach is consistent with the theory of LC, while inheriting its limitations—that is, LC guarantees accuracy only before 70% of a vector is exhausted [37]. After a sampling event, the vector is recycled (reset to 0) in anticipation of the next round sampling event (⑦). By doing so, the reduction ratio of each flow (equivalently, the sampling rate) is approximately $1/\hat{p}$. Due to the constrained memory space, however, vectors have to be designed to share bit positions with one another (virtual vector hereafter), which brings about a major challenge, the noise owing to virtual vector collisions. That is, multiple concurrent flows fall in a race condition when claiming bits in shared bit positions. We carefully designed the sketch to take a noise-free approach by minimizing the race condition (See section III-C).

B. Decoding: Sampling Trigger

Understanding the Saturation Event. A sampling event is triggered by the saturation of a virtual vector assigned to each flow, and the usage of virtual vector is monitored whenever a packet is encoded into it. The packet that triggers the saturation event of a vector (hereafter, signaling bit) is one that flips a 0's position to 1 and eventually causes more than 70% usage of the vector. We observed that the LC's formula could not be directly applied here, because it overestimates the number of packets encoded in the virtual vector. The key reason is that the last packet in a 70% marked vector is highly likely to remark the bit already marked in LC, whereas the signaling bit marks a fresh 0's bit in our sketch. We note that this estimation gap does not mean that LC is wrong, but event-driven sampling trigger (*i.e.,* signaling bit) was not intended by LC.

Saturation Event-based Estimation (Sampling Interval). Here, we propose a new formula to calculate the estimation considering the saturation event, which is the basis of the real-time sampling.

Theorem 1: Considering the saturation event that triggers setting the $(s-z+1)$-th signaling bit in a virtual vector of size s, the sampling interval of a flow, $\hat{p}$ is calculated as follows:

$$\hat{p} = \frac{\ln V_z}{\ln\left(1-\frac{1}{s}\right)} + \left(\frac{1-V_{s-z}^{\tau}}{1-V_{s-z}} - \tau \cdot V_{s-z}^{\tau}\right), \tag{1}$$

where z (V_z) is the fraction of 0's in a virtual vector, τ is a positive constant, and s is the vector size. For convenience, we consider the first term as $f(z)$ and the second as $g(z)$.

Proof: The equation consists of two parts: the former modifies the LC's formula without truncating the minor terms, and the latter is the probabilistic expectation by considering the saturation event. Let n be the number of packets and $\hat{n}$ be the estimation of packets. In appendix A in [37], Wang *et al.* derived the mean of the random variable U_n which represents the number of 0's in the bit map, or a virtual vector. Let A_j be an event that the j-th bit is 0, and let 1_{A_j} be the corresponding indicator random variable. Then, since U_n is the number of 0's, $U_n = \sum_{j=1}^{s} 1_{A_j}$, where s is the size of the vector. Finally, per [37], we have the following.

$$E(U_n) = \sum_{j=1}^{s} P(A_j) = s \cdot (1-1/s)^{\hat{n}}, \tag{2}$$

where $P(A_j)$ is the probability of A_j. Since the assignment of the bits is independent, $P(A_j) = (1-1/s)^{\hat{n}}$.

They approximate this equation to a convergence value when s and n go to infinity. However, for a more precise estimation, Nyang *et al.* [29] used the non-approximation estimation derived from the expectation of U_n, which is used as $f(z)$ for better accuracy, because the frequent accumulation of small estimation error can grow bigger. They obtained

$$V_z = (1-1/s)^{\hat{n}}, \tag{3}$$

where V_z is the fraction of 0's in the vector, that is, $E(U_n)/s$. And by taking the log, they deduced

$$\ln V_z = \hat{n} \cdot \ln(1-1/s). \tag{4}$$

We choose $\hat{n}$ as $f(z)$, because $\hat{n}$ is the estimation of packets when there are z zeros in the vector. Note that the first part $f(z)$ is not a cumulative sum of the second part $g(z)$ because z is the number of zeros before the signaling bit flips to 1's.

Let $g(z)$ be the expected number of packets required for saturation after a virtual vector state reaches to the state having $z-1$ zero bits from z zero bits. We assume that $g(z)$ is the number of packets needed to make the event. This means that the first $g(z)-1$ packets did not convert a new 0-bit to 1, and the last $g(z)$-th packet selects the 0-bit in the virtual vector. The probability of the former is V_{s-z}, the fraction of 1's in the virtual vector v, and the latter V_z, the fraction of 0's in v; namely, V_z equals $1-V_{s-z}$. Since $g(z)$ must be a positive integer, we can expect $g(z)$ from 1 to some extent, τ. Therefore, we get the following expectation:

$$\begin{aligned} g(z) &= V_z + 2V_zV_{s-z} + 3V_zV_{s-z}^2 + \cdots + \tau V_zV_{s-z}^{\tau-1} \\ &= \sum_{i=1}^{\tau}\left(iV_zV_{s-z}^{i-1}\right) = \frac{1-V_{s-z}^{\tau}}{1-V_{s-z}} - \tau \cdot V_{s-z}^{\tau}. \end{aligned}$$

The last term in the above equation is obtained from $g(z) - V_{s-z} \cdot g(z)$. V_z in $g(z)$ is canceled out by dividing both sides by V_z; that is, $1-V_{s-z}$. ■

In this paper, we set the number of trials (τ) to 8 because it has 95% of confidence on flipping a new 0's to 1's from having z zeros. A random variable $\mathcal{K}$ follows the binomial distribution with parameters τ and V_z, where τ is the number of trials (or packets) and V_z is a probability that one packet make the saturation event.

Proof of Unbiased Sampling. The first term is unbiased when it is used to estimate the average number of packets per the virtual vector usage (See [37]). We use it to estimate the condition before the saturation event (*i.e.,* $f(z)$). The second term is the expected number of packets (constant) that triggers the saturation event from the last condition (*i.e.,* $g(z)$), which does not impact the variance of the entire formula.

Theorem 2: Assume that there is an initial virtual vector v for SketchFlow. We define the saturation event by the state transition from the state where the number of zeros in v is z to the state with $z-1$ (z is 30% of the virtual vector size when $s \geq 8$.). At the exact moment when the event has just occurred, SketchFlow's estimation of the number of packets needed to trigger an event is unbiased.

Proof: The first term of the estimation, $f(z)$, is the number of packets which is used to maintain z zeros in v. The expected value of $f(z)$, $E(f(z))$, is unbiased by LC's theory. Starting from the point when v has z zeros, the expected value of the number of packets for the saturation event is $E(g(z))$, which is also unbiased according to Theorem 1. Therefore, SketchFlow's formula $f(z)+g(z)$ is unbiased, because $E\left(f(z)+g(z)\right) = E(f(z)) + E(g(z))$. ■

C. Estimation without Noise Reduction

In SketchFlow, a fixed virtual vector (of s bits) was "temporally" given to a flow for performing LC-like probabilistic counting. Thus, vectors of concurrent flows may partially or fully share bit positions, and bring about a race condition for the shared bit positions resulting in a virtual vector collision. We propose a noise-free approach to dramatically mitigate the virtual vector collision spatially and temporally. We also show that even when the noise occurs, SketchFlow can ignore the vector collision problem introducing the noise. For instance, once a specific flow triggers saturation event of the virtual vector, the flow takes all bits in the vector regardless of how many bits (or noises) were actually contributed by other flows, and it resets the vector. Our approach is tolerant to collision considering the following dispersion aspects:

Spatial Dispersion. Spatially, SketchFlow confines the virtual vector of flows within a word range (*i.e.,* 32-bit or 64-bit), then distributes flows in the memory space (*i.e.,* word array) uniformly. This greatly reduces the probability of collision of concurrent flows, when enough number of words for confinement are given. In a local view, SketchFlow uses a small size for virtual vectors, which is smaller than the word size. The probability of vector collision within a s_w-bit word with respect to the size of vector (s_v) and the number of concurrent flows (n_f) is $p_{collision} = 1/\binom{s_w}{s_v \cdot n_f}$, where $p_{collision}$ decreases when s_v gets smaller. Both contribute to reducing spatial collision of virtual vectors of concurrent flows.

Temporal Dispersion. SketchFlow looks into a small timescale for TCP bursts. TCP usually sends a window of data in one or a few bursts and waits for ACKs, which causes a flow to be broken into many small subsets named flowlets. Sinha *et al.* [33] reported that the number of concurrent flowlets was much smaller than that of concurrent flows, which makes the probability of the spatial virtual vector collision even smaller in the smaller timescale. Moreover, the small vector size of SketchFlow increases the probability that the saturation events are triggered before the end of flowlets, which also reduces the probability of virtual vector collision in a temporal manner.

Worst Case. For the worst case, we can consider the situation where multiple concurrent flowlets share bit positions with each other. We claim that even without considering the noise by other concurrent flowlets, equation (1) is enough to decide whether the flow reaches the sampling interval or not. Whether two flows are mouse or elephants, probabilities of each flow to lose bits are the same in a sampling interval. This is because, during a sampling interval, two flows lose the concept of transmission rate but are only mixed in a random sequence in the buffer when concurrently arriving flowlets are loaded.

D. Scalable Sampling

As described in section III, SketchFlow uses a virtual vector smaller than the size of the word. However, a 32-bit virtual vector cannot count over 40 (See Fig. 4(a)), which limits the minimum sampling rate. Increasing the confinement size does not help with scaling up the sampling interval but induces more memory read and write. To scale up the sampling interval, SketchFlow employs a "multi-layer" strategy where each layer of SketchFlow maintains an independent word array. Unlike other multi-layer sketch approaches that only scale up the retention capacity (e.g., [3]), SketchFlow provides an online decoding feature as well to help with the high-speed processing. Encoding the arriving packet starts from the lowest layer and climbs the layers depending on the saturation of the virtual vector. Repeatedly, the saturation from the lower layer is encoded into its upper layer following the same process of encoding. That is, the upper layer counts the saturation of its lower layer. Finally, the sampling event happens when the flow is saturated at the highest layer. All layers share the same hash value of a flow but run different random functions. The sampling interval of multi-layer SketchFlow is the multiplication of the sampling interval of each layer (See Fig. 4(b) for sampling interval by different layers). For 3-layer SketchFlow with an 8-bit virtual vector, the sampling interval is 9.764^3. Note that each layer can use different virtual vector sizes to achieve different sampling intervals on demand.

Algorithm 1: Encoding and Sampling Trigger

```
input: # of layer $l$, word array $w[l][]$ , vector size $s$
1  forall $Pkt_f$ do
2      $h_f \leftarrow$ hash($Pkt_f$);
3      $w_v \leftarrow$ make_confined_vector($h_f$);
4      for $L$ = 0 to $l-1$ do
5          $w[L][h_f] \leftarrow w[L][h_f]$ | leave_one_bit_only($w_v$);
6          /*Saturation event is triggered if usage > 70%*/;
7          if Popcount($w[L][h_f]$ & $w_v$) $\geq 0.7 \times s$ then
8              $w[L][h_f] \leftarrow w[L][h_f]$ & bitwiseNOT($w_v$);
9              /*Sampling event is triggered in the last layer*/;
10             if $L = l-1$ then
11                 Trigger a sampling event with flow $f$;
12             end
13         else
14             break;
15         end
16     end
17 end
```

IV. Implementation

A. Algorithm

SketchFlow's algorithm can be divided into encoding, sampling/saturation trigger, and multi-layer sampling phases.

Encoding. For each arriving packet of a flow f, SketchFlow computes the hash (h_f) of the 5-tuple extracted from the header (line 3). The h_f is used for two purposes. First, part of h_f is used to calculate the bit positions of the virtual vector in a word (line 4). By calling `make_confined_vector()`, we obtain a virtual vector bit mask in a word register (w_v) for one confinement in which only the bit positions of the virtual vector for f are set. Second, h_f is regarded as an index that determines in which word the virtual vector is confined among word arrays (line 5). Once w_v and $w[Layer][h_f]$ are ready, `leave_one_bit_only()` randomly selects one of the 1's position among w_v and "Bitwise OR" it with $w[Layer][h_f]$.

Sampling/Saturation Trigger. After several rounds of encoding, the virtual vector of f will be saturated (>70% usage). SketchFlow monitors the saturation of the vector after every encoding by counting the number of 1's using `Popcount()` [25] (line 7). Once the saturation threshold is reached, the bit positions will be reset to 0 (line 8), and the sampling/saturation event is triggered[1]. One sampled packet represents $\hat{p}$ packets in equation (1), which is a pre-decoded value and enables real-time sampling.

Multi-layer Sampling. To implement multi-layer SketchFlow, the encoding process is repeated (line 4-16) for each saturation event to the upper layer using the word array the layer belongs to. Eventually, the sampling event is triggered when the saturation events occur in the last layer (line 11-12). All layers share the hash value (h_f) and virtual vector mask(w_v) computed in the lowest layer to alleviate the computation.

B. Parameter

The size of confinement of a virtual vector is selectable depending on the processor architecture (32 or 64 bits). The size of the virtual vector is recommended not to exceed half of the size of a word to reduce the probability of virtual vector collisions within a word. For memory usage, we recommend that the maximum possible number of virtual vectors that can be contained in a word array should be equivalent to the number of concurrent flows in a second for tolerant sampling. In our evaluation, we used a 110KB 32-bit word array per layer and an 8-bit virtual vector when performing the experiments using CAIDA trace because the maximum number of concurrent flows was $\approx$110K. We found that SketchFlow provides better accuracy than other sketch approaches even with a small memory usage (See section V-E).

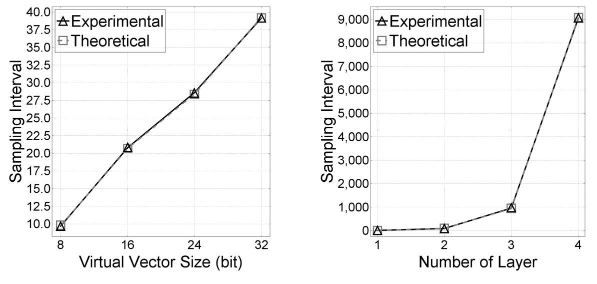
C. Performance Optimization

For real-time per-flow systematic sampling, we take several optimization efforts. 1) By careful design, SketchFlow requires only one conditional branch for each layer to trigger the sampling/saturation event. 2) For fast computation, SketchFlow marks the bit positions of the virtual vector in an empty register (line 3) so that encoding (line 5) and recycling (line 8) can be done in a single "Bitwise OR" and "Bitwise AND" operations. 3) Due to the confinement of a virtual vector, usage check of a virtual vector can be done using a built-in hardware population counting function (`Popcount()`) [25]. 4) Inspired by the implementation of the exact match cache (EMC) module of OpenvSwitch using DPDK [6], the hardware-based CRC checksum instruction of streaming SIMD extensions (SSE) [34] was used to calculate our 5-tuple hash function.

V. Evaluation

In this section, we use various metrics to evaluate SketchFlow. First, we compare our theoretically-estimated sampling interval with the experimental result to verify the sampling interval in equation (1). Also, we show the scalability of our multi-layer strategy in terms of the sampling interval. Second, we evaluate the overall performance of SketchFlow using CAIDA trace by varying the sampling rate and comparing SketchFlow with simple random sampling (sFlow [32]) and with a non-linear scheme (sketch guided sampling [19], SGS hereafter). Third, we discuss the overhead of SketchFlow. Lastly, we evaluated SketchFlow not only in the network traffic dataset [1] but also in the keyword ranking problem (Twitter dataset [20]) and in the hot block ranking problem (Disk I/O trace [27]) that has more complex data distribution (iv).

[1]If $s = 8$, a sampling event is triggered when 6 or more 0's positions are marked as 1's (*i.e.,* $k = 6$), because 6 bits are 75% (>70%) of an 8-bit vector.



(a) Single-layer design (b) Multi-layer design

Fig. 4. Theoretical and experimental sampling interval of SketchFlow.

A. Estimation Accuracy and Scalability

Fig. 4(a) shows sampling interval of SketchFlow by varying the virtual vector size. The Y-axis is the average number of packets to trigger a sampling/saturation event. We compared the estimated value of SketchFlow with the experimental results (1 million runs). As a result, our estimation is accurate regardless of the size of the virtual vector (error rate<0.07% for 8-bit). However, the growth rate is very slow, and so the counting capacity for a 32-bit virtual vector cannot go over 40 packets (Fig. 4(a)). With the multi-layer strategy, the counting capacity exponentially increased, as shown in Fig. 4(b). Using an 8-bit virtual vector for 4-layer SketchFlow which equally assigned 32 bits for each flow, the counting capacity dramatically increased to reach around 9,088. Note that to achieve the equivalent counting capacity without the multi-layer strategy, thousands of bits are needed for a virtual vector. Furthermore, hundreds of memory accesses are required to decode it, which is unacceptable for online sampling. In the multi-layer mode, SketchFlow needs only one memory access for each layer.

B. SketchFlow vs. Linear Sampling Approach (SRS)

Per-flow Accuracy. For our baseline, we compared SketchFlow with SRS using the CAIDA trace. The implementation of SRS followed the way used in sFlow. To achieve the same sampling rate as SRS, SketchFlow approximated the sampling rate using the multi-layer strategy where each layer used 8-bit virtual vector. The approximated sampling rates of SketchFlow are $1/9.764$ (L1), $1/95.328$ (L2), $1/930.750$ (L3) and $1/9087.749$ (L4), respectively. In SketchFlow, each layer was assigned with a 110KB 32-bit word array so that the maximum possible number of virtual vectors without collision should be equivalent to the maximum concurrent flows of CAIDA trace in a second. No memory usage is required by SRS. Fig. 5 presents the relative error of SketchFlow and SRS varying sampling rates, where SketchFlow's estimation is unbiased from the ground truth and its accuracy is better than SRS's, regardless of flow sizes. Also, SRS's variance grows faster as the sampling rate decreases. Fig. 6 shows the CDFs

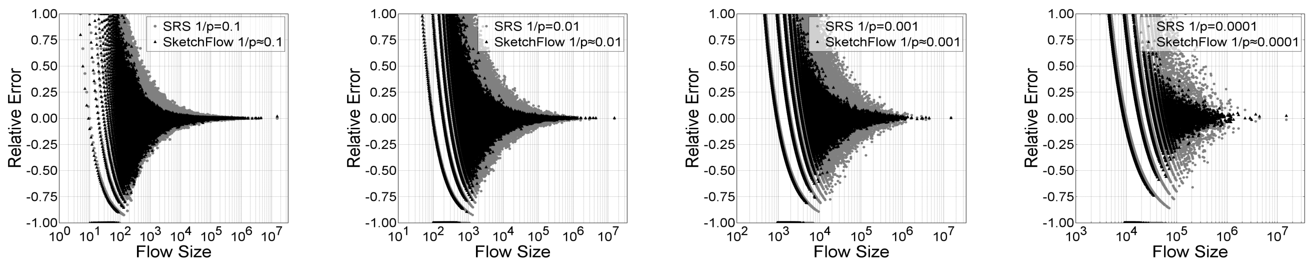


Fig. 5. CAIDA trace: Relative error of independent flows of SketchFlow and SRS. Each point stands for each flow. To see how accurate each scheme is, check how close the point is to $y = 0$. Multi-layer SketchFlow was used to approximate sampling rates 0.01-0.0001 (left to right), respectively. Each layer was assigned with a 110KB 32-bit word array, and 8-bit virtual vector was used for all experiments. No memory usage is required by SRS. CAIDA trace contains ≈2 billion packets and ≈95 million L4 flows.

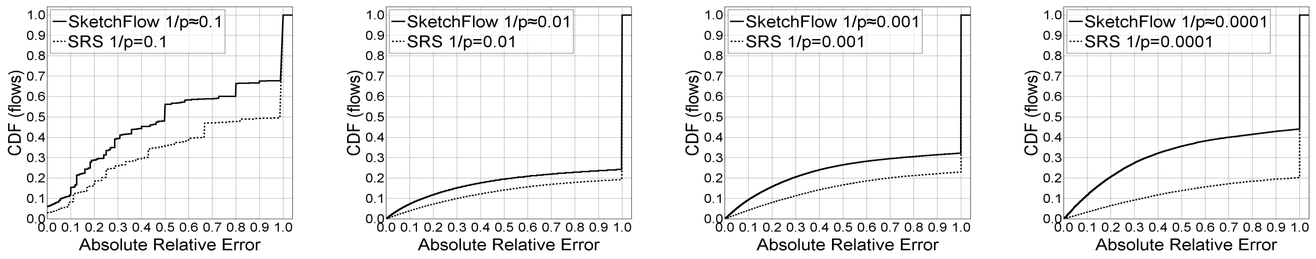


Fig. 6. CAIDA trace: CDF of flow-level relative error of SketchFlow and SRS. The overall accuracy of SketchFlow is better than SRS.

TABLE I
FLOW THINNING PERFORMANCE: HIGHER IS BETTER

Sampling Rate		0.1	0.01	0.001	0.0001
SketchFlow	precision	0.414	0.174	0.240	0.293
	recall	0.931	0.950	0.959	0.954
SRS	precision	0.408	0.161	0.201	0.159
	recall	0.916	0.960	0.921	0.923

TABLE II
PACKET THINNING PERFORMANCE

Sampling Rate	SketchFlow		SRS	
	samples	ratio	samples	ratio
0.1	198,322,728	0.10156	195,274,392	0.10000
0.01	19,973,488	0.01023	19,531,764	0.01000
0.001	1,964,032	0.00101	1,952,120	0.00100
0.0001	154,041	0.00008	195,865	0.00010

of overall flow-level relative error of both schemes according to the sampling rate. Both were compared with the ground truth. As shown, SketchFlow is more accurate than SRS in all cases where the sampling rates ranged from 0.1 to 0.0001.

Flow Thinning. We evaluated the quality of flow thinning (sampling). Precision refers to the fraction of correctly-sampled flows (*i.e.,* sampled flows where the size is equal to or greater than the sampling interval) over all sampled flows. As shown in Table I, the overall precision of SketchFlow is higher than that of SRS. The precision gap is even greater when the sampling rate decreases. The recall is the fraction of correctly-sampled flows over flows that are supposed to be sampled (*i.e.,* all the flows of which sizes are equal to or greater than the sampling interval). As a result, the recall of SketchFlow is shown to be better than SRS in most cases. Overall, the quality of SketchFlow in flow sampling is better than or equal to that of SRS. Note that when the sampling rate is 0.01, the precision is low in comparison with 0.1 and 0.001 due to the drastically-increased mouse flows.

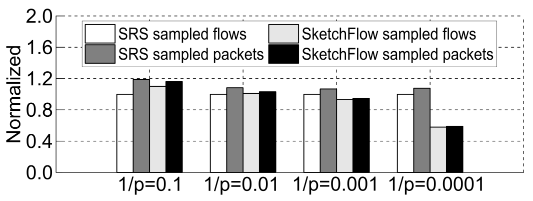
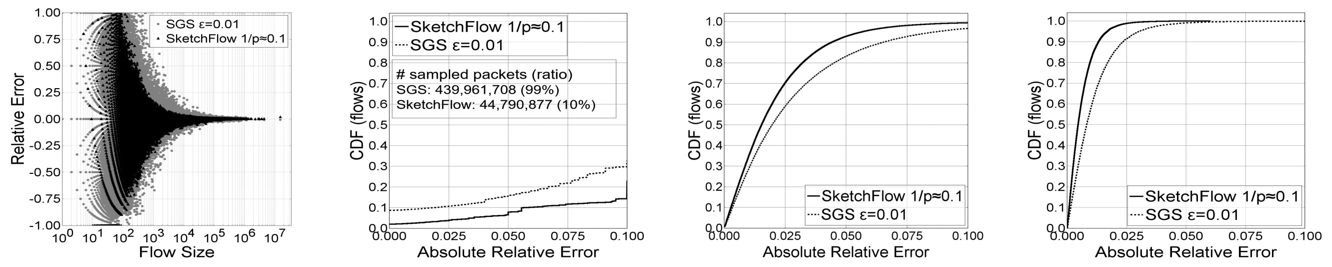


Fig. 7. Comparison of mouse flow sampling between SketchFlow and SRS. Mouse flow is a flow which the volume is less than sampling interval p

Packet Thinning. We compared the fraction of the sampled packets over the entire packets of the CAIDA traffic. As shown in Table II, SketchFlow guarantees the traffic reduction rate, which can relax the overhead under a fixed boundary.

Mouse Flow Sampling. One of the desirable features of SRS is the ability to provide mouse flow samples. The mouse flow is referred to a flow of which the volume is less than the sampling interval p. We note sampling of mouse flows is irrelevant to the size of the flow, which means one-packet sized mouse flows also have a chance to be sampled because of noise in the virtual vector. Fig. 7 shows a comparison between SketchFlow and SRS with respect to the number of sampled flows and the sampled packets. As shown, SketchFlow captures comparable or more mouse flow samples than SRS with sampling rates ($1/p$) of 0.1, 0.01, and 0.001. This illustrates that SketchFlow can be a good alternative to SRS for general-purpose sampling tasks without losing the information of mouse flows, but providing better accuracy of elephant flows. Unsurprisingly, though, when the sampling rate is 0.0001, the number of the sampled mouse flows is halved compared to SRS. This is because SketchFlow uses a sketch saturation-based sampling mechanism. Since our dataset follows a heavy-tailed distribution [2], the volume increment of mouse flows following the increment of the sampling interval (p) is slow. Thus, it is hard for mouse flows to saturate the sketch for



(a) Relative Error (b) Small Flows (10-1,000) (c) Medium Flows (1,000-10,000) (d) Large Flows (10,000+)

Fig. 8. CAIDA trace: Accuracy comparison between SketchFlow and SGS. Both were assigned with 110KB memory for fair comparison. The sampling rate of SketchFlow was 0.1 and the expected relative error of SGS was 0.01. (a) shows the relative error of independent flows. Each point stands for each flow. The closer point to $y = 0$, the better accuracy. (b)-(d) show the CDF of relative error of different flow size intervals.

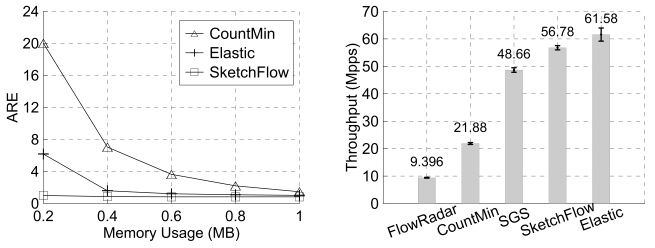
triggering sampling events. We note that the efficiency of mouse flow sampling of SketchFlow is better than that of SRS with any sampling rate, which means SketchFlow can capture more mouse flows with fewer samples.

C. SketchFlow vs. Non-linear Sampling Approach (SGS)

We compared SketchFlow with a non-linear scheme, SGS [19]. For fairness, both SketchFlow and SGS used 110KB memory space for their sketch. As shown in Fig. 8(a), the overall relative error of SketchFlow is closer to 0 than SGS's by varying the flow size. Remarkably, SGS outperforms SketchFlow in small flows (Fig. 8(b)) but not large flows (Fig. 8(c)-(d)). The result is reasonable and anticipated because the strategy of SGS is to sample mouse flows with a very high probability, which leads to the frequent sampling of mouse flows. Fig. 8(b) shows that SketchFlow samples only 10% (44M packets), compared to what SGS sampled (440M packets). The estimation of SGS is accurate, and it guarantees the relative error of most flows is within the expected margin (ϵ = 0.01 in our experiments). However, the most critical problem of SGS is packet thinning: in our experiments, SGS triggered 53% sampling events over the entire traffic because a large number of mouse flows appear in the CAIDA trace, the real-world dataset. This unacceptably high sampling rate explains the impracticality of SGS as well as the high accuracy for mouse flows. Unlike SGS, in terms of the flow table overhead (NetFlow) or the network overhead (sFlow), SketchFlow guarantees the desired overhead relaxation rate than SGS.

D. SketchFlow vs. Sketch Approaches

We further compared SketchFlow with three state-of-the-art sketch approaches: CountMin [5], Elastic sketch [38] and FlowRadar [22]. We followed experiments in Elastic sketch [38] and divided a one-hour CAIDA dataset into 720 five-second subset traces. We varied the memory usage from 0.2MB to 1MB and evaluated the accuracy in terms of the average relative error ($ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|f_i - \hat{f}_i|}{f_i}$). For Elastic sketch, we fixed the heavy-part with 150KB memory and the remaining for the light-part. For CountMin, we used 3 hash functions as recommended in [11]. In Fig. 9(a), we found that SketchFlow achieves the lowest ARE in all cases, while using similar or even using less memory. On the contrary, accuracy degradation is observed for both CountMin and Elastic sketch



(a) Accuracy Comparison (b) Speed Comparison

Fig. 9. SketchFlow vs. Sketch Approaches: comparison of memory usage, accuracy and processing speed of sketches on a CPU platform.

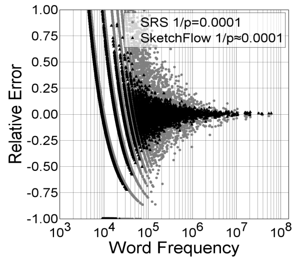
following the decrease in memory usage. We also conducted the same experiment using FlowRadar, which failed to decode any flow using 1MB memory.

E. Overhead Evaluation

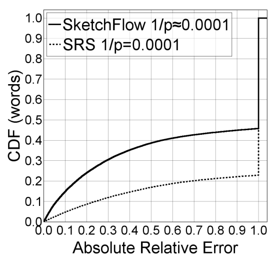
To evaluate the overheads, we conducted experiments with a testbed that is equipped with Xeon E5-2620 v4 2.10GHz, which supports Streaming SIMD Extensions (SSE).

CPU Platform. We evaluated SketchFlow in terms of throughput (Mpps) using a CPU platform. We compared our approach with four solutions (FlowRadar, CountMin, Elastic sketch, and SGS). As shown in Fig. 9(b), SketchFlow achieved higher throughput than FlowRadar, CountMin, and SGS. SGS can reach a throughput of 48.66 Mpps, while SketchFlow is 1.16 times faster (*i.e.,* 56.78 Mpps). Remarkably, Elastic achieved the highest throughput (*i.e.,* 61.58 Mpps), which is 1.08 times faster than SketchFlow. However, we note that we did not involve any sketch or sample sending in this experiment. Elastic sketch requires a sketch compression process for saving bandwidth overhead caused by sketch delivering.

OpenvSwitch. To comparatively evaluate the overhead of SketchFlow, we integrated SRS (sFlow) and SketchFlow in the packet processing pipeline of OpenvSwitch (using DPDK 17.11.2 [6]). We generated the CAIDA trace using Intel X540AT2 10G NIC and `pktgen` [6] for measuring the average cycles required to make the sampling decision of a packet. In this experiment, a 4-layer SketchFlow was used to approximate the sampling rate of 0.0001 to compare with SRS ($1/p$ = 0.0001). According to the experimental results, SRS required fewer cycles (52 cycles/packet), and SketchFlow required slightly more than SRS; 69 cycles per packet. When comparing SRS with SketchFlow, the additional hash computation

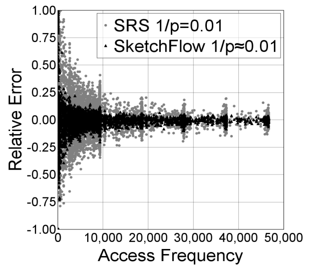


(a) Relative Error

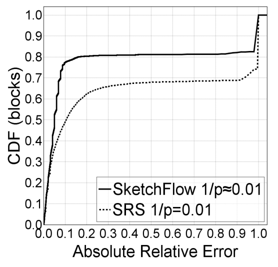


(b) CDF

Fig. 10. Twitter dataset: Accuracy of SketchFlow and SRS. Both were evaluated with sampling rate 0.0001. Tweet dataset contains ≈7 billion subunits including word, link, name, *etc*.



(a) Relative Error



(b) CDF

Fig. 11. Disk I/O trace: Accuracy of SketchFlow and SRS. Both were evaluated with sampling rate 0.01. Disk I/O trace contains 170 million I/O requests of 390 thousand different offsets.

overhead of SketchFlow is large, although can be substantially reduced using hardware-based functions (*i.e.,* CRC instruction of SSE), and a few memory accesses are also acceptable for online sampling. Through an in-depth examination, we found that the overhead of SketchFlow occurred mostly in calculating the bit positions of the virtual vector of flows. This overhead can be mitigated by caching the virtual vector of the last flow because a frequent burst behavior of the same flow has been observed in many modern traffic loads [16].

F. Twitter and Disk I/O trace

We also examined the scalability of SketchFlow using a large dataset (Twitter dataset) and its versatility using a dataset with a different distribution (Disk I/O trace). As results, SketchFlow outperforms SRS for both datasets in terms of the relative error. For the Twitter dataset, we used the sampling rate of 0.0001 by considering its scale. As shown in Fig. 10(b), the overall absolute relative error of SketchFlow is much smaller than SRS. Moreover, SketchFlow is shown more accurate than SRS for different word frequencies, and the variance of SketchFlow is much smaller (Fig. 10(a)). While the scale of disk I/O trace is much smaller than Twitter's, it presents a different distribution. A sampling rate of 0.01 was used reflecting the fact that most of the blocks were accessed under 10^5 times. As shown in Fig. 11, SketchFlow performs better than SRS in terms of the relative error and variance.

VI. Related Work

Sampling is implemented using one of two approaches: timer- and packet-driven sampling. Timer-driven sampling is chosen by both sFlow [32] and NetFlow [8], [28]. However, the packet-driven approach is preferred in practice because of its performance. Therefore, several packet-driven approaches have been proposed since its introduction, initially to measure the NSFNET backbone. Claffy *et al.* described three different sampling methods, simple random sampling, stratified sampling, and systematic sampling [4]. Hohn and Veitch [13] compared packet-level sampling's inaccuracy over flow-level sampling's. Duffield *et al.* [7] argued that flow-level sampling is unstable under resource constraints and proposed a threshold-based sampling. In both works, the flow sampling schemes showed higher accuracy than the packet sampling. However, the traffic reduction rate cannot be guaranteed [19].

Another line of works used non-linear sampling rates. Kumar *et al.* [19] introduced a non-linear scheme (SGS) using different probabilities depending on the size of the flows. Their approach acknowledges that information on mouse flows is likely to be lost using a linear approach. SGS employed a compact sketch to record the flows' size with a higher probability for smaller flows. Hu *et al.* [14] and Ramachandran *et al.* [30] introduced similar approaches with different architectures and data structures, providing high accuracy in flow size estimation with mouse flows. However, the high sampling probability of mouse flows leads to a huge number of samples, negatively affecting the traffic reduction rate.

VII. Conclusion

In this paper, we introduced a new notion of per-flow systematic sampling, where the sampling accuracy is shown to be superior to that of the simple random sampling. To realize this idea, we proposed a new sampling framework using sketches as per-flow samplers. In this framework, a per-flow sketch saturation event works as a signal to sample a packet in a flow, and the per-flow saturation interval as the per-flow sampling interval. Instead of using a sketch as a full flow size estimator that necessarily causes sketch saturation and offline decoding, we had our new sketch algorithm measure only the sampling interval and be emptied for reuse in real-time. With this framework and a sketch algorithm, we successfully built a highly-accurate sampling algorithm, SketchFlow, which is able to perform per-flow systematic sampling. We showed proof on SketchFlow's accuracy and demonstrated performance by experiment with real-world datasets such as traces from the network, I/O, and social network platforms. We believe that our work opens a new direction in data sampling, and we expect that SketchFlow would inspire more work on per-flow sampling.

Acknowledgement

This research was supported by Grobal Research Laboratory (GRL) Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2016K1A1A2912757).This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1A2B4010657). DaeHun Nyang is the corresponding author.

References

[1] The cooperative association for internet data analysis, equinix chicago data center. https://www.caida.org. [13:00-14:00, Apr 19 2018., from Sao Paulo to New York,].

[2] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 267–280, 2010.

[3] M. Chen, S. Chen, and Z. Cai. Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE/ACM Trans. Netw.*, 25(2):1249–1262, 2017.

[4] K. C. Claffy, G. C. Polyzos, and H. Braun. Application of sampling methodologies to network traffic characterization. In *Proceedings of the ACM SIGCOMM '93 Conference on Communications Architectures, Protocols and Applications, San Francisco, CA, USA, September 13-17, 1993*, pages 194–203, 1993.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[6] Data Plane Development Kit. https://www.dpdk.org/.

[7] N. G. Duffield, C. Lund, and M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Trans. Information Theory*, 51(5):1756–1775, 2005.

[8] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA*, pages 245–256, 2004.

[9] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 19-23, 2002, Pittsburgh, PA, USA*, pages 323–336, 2002.

[10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[11] A. Goyal, H. D. III, and G. Cormode. Sketch algorithms for estimating point queries in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 1093–1103, 2012.

[12] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[13] N. Hohn and D. Veitch. Inverting sampled traffic. *IEEE/ACM Trans. Netw.*, 14(1):68–80, 2006.

[14] 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 Trans. Communications*, 60(3):789–798, 2012.

[15] 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*, pages 113–126, 2017.

[16] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of the ACM Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, pages 133–138, 2013.

[17] 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*, pages 177–188, 2004.

[18] A. Kumar, J. Xu, and J. Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12):2327–2339, 2006.

[19] A. Kumar and J. J. Xu. Sketch guided sampling - using on-line estimates of flow size for adaptive data collection. In *Proceedings of the 25th IEEE International Conference on Computer Communication, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2006, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[20] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[21] T. Li, S. Chen, and Y. Ling. Fast and compact per-flow traffic measurement through randomized counter sharing. In *Proceedings of the 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2011, 10-15 April 2011, Shanghai, China*, pages 1799–1807, 2011.

[22] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 311–324, 2016.

[23] P. Lieven and B. Scheuermann. High-speed per-flow traffic measurement with probabilistic multiplicity counting. In *Proceedings of the 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2010, 15-19 March 2010, San Diego, CA, USA*, pages 1253–1261, 2010.

[24] 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*, pages 101–114, 2016.

[25] Hamming weight. https://software.intel.com/sites/landingpage/ IntrinsicsGuide/#text=_mm_popcnt_u32.

[26] R. Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978.

[27] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. *TOS*, 4(3):10:1–10:23, 2008.

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

[29] D. Nyang and D. Shin. Recyclable counter with confinement for real-time per-flow measurement. *IEEE/ACM Trans. Netw.*, 24(5):3191–3203, 2016.

[30] A. Ramachandran, S. Seetharaman, N. Feamster, and V. V. Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference, IMC 2008, Vouliagmeni, Greece, October 20-22, 2008*, pages 257–270, 2008.

[31] Sandvine. Global internet phenomena report. 2016.

[32] sFlow. http://www.sflow.org/.

[33] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *ACM HotNets*, 2004.

[34] Intel SSE4 Programming Reference. https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf.

[35] P. Tune and D. Veitch. Towards optimal sampling for flow size estimation. In *Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference, IMC 2008, Vouliagmeni, Greece, October 20-22, 2008*, pages 243–256, 2008.

[36] P. Tune and D. Veitch. Sampling vs sketching: An information theoretic comparison. In *Proceedings of the 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2011, 10-15 April 2011, Shanghai, China*, pages 2105–2113, 2011.

[37] K. Whang, B. T. V. Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.

[38] 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*, pages 561–575, 2018.

[39] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 29–42, 2013.