Routing Oblivious Measurement Analytics

Ran Ben Basat, Harvard University
Xiaoqi Chen, Princeton University
Gil Einziger, Ben Gurion University
Shir Landau Feibish, Princeton University
Danny Raz, Technion
Minlan Yu, Harvard University

***Abstract*—Network-wide traffic analytics are often needed for various network monitoring tasks. These measurements are often performed by collecting samples at network switches, which are then sent to the controller for aggregation. However, performing such analytics without "overcounting" flows or packets that traverse multiple measurement switches is challenging. Therefore, existing solutions often simplify the problem by making assumptions on the routing or measurement switch placement.**

We introduce AROMA, a measurement infrastructure that generates a uniform sample of packets and flows regardless of the topology, workload and routing. Therefore, AROMA can be deployed in many settings, and can also work in the data plane of programmable PISA switches. AROMA includes controller algorithms that approximate a variety of essential measurement tasks while providing formal accuracy guarantees. Using extensive simulations on real-world network traces, we show that our algorithms are competitively accurate compared to the best existing solutions despite the fact that they make no assumptions on the underlying network or the placement of measurement switches.

I. Introduction

Many network applications, such as load balancing, QoS enforcement, intrusion detection, and traffic engineering [23], [6], [9], [14], [15], [21], [22], [27], [31], [32], rely on network-wide analytics of the network traffic to reach informed decisions. Network-wide analytics is often done by collecting samples of traffic from individual switches [33]. These samples are then sent to the controller, where the data from all measurement switches is combined to assemble a network-wide view of the traffic. Such samples can approximate a variety of essential measurement tasks such as identifying the heavy hitter flows [8], calculating hierarchical heavy hitters [5], [41], estimating the flow size distribution, and identifying super-spreaders and port scans [26], [40].

Samples may be collected at either *flow* or *packet* granularity. In *packet sampling*, each measurement device samples a packet with probability p. *Flow sampling* [17], [34], [35] is a more complex primitive, where each measurement device samples packets based on their flows; however all flows in the network should be sampled with *the same* probability, regardless of their size. Inherently, packets that traverse multiple devices, or heavier flows, are more likely to be sampled, therefore skewing the network-wide measurements. This raises the need for uniform sampling techniques. In uniform sampling, any packet has an equal chance of being included in the globally collected sample regardless of the number of measurement switches it goes through (as long as it traverses at least one measurement switch). Similarly, any flow has an equal chance of being sampled regardless of the number of packets in the flow.

ISBN 978-3-903176-28-7

To achieve uniform sampling, existing works often assume that packets only traverse a single device [24], [42]. However, this restricts the measurement device positioning and may only work for specific routing protocols and network topology (e.g., fat trees). Other solutions rely on packet marking to ensure that each packet is considered only once [3]. This technique can be easily exploited by adversaries (e.g., an attacker can mark all its packets to avoid detection). Alternatively, some solutions make non-trivial assumptions abut the underlying network characteristics, such as network topology, routing protocols, and traffic dynamics. For example, cSamp [34] requires per-switch configuration, and a traffic matrix detailing the number of flows between each source and destination. While these requirements may be sustainable for some networks, they may not be reasonable in large, rapidly changing networks.

In this paper, we present AROMA (**A**pproximate **R**outing **O**blivious **M**easurement **A**nalytics), a measurement analytics infrastructure that collects uniform flow and packet samples in any network topology. AROMA is workload and routing oblivious, so packets can traverse multiple measurement devices without biasing the sample. The underlying technique used in AROMA is a k-partition hash based structure, which supports sampling based on the packet or flow identifier. Therefore, AROMA can be configured to perform packet sampling, flow sampling, or both. The main features of AROMA are:

Routing and workload oblivious measurements. AROMA makes no assumptions about the network topology or routing and makes no packet modifications. Measurement switches produce samples without any need for coordination between them (e.g., there is no need to tag a packet[3]) and without per-switch configuration (i.e., there is no need to divide responsibility for sampling parts of the traffic across the measurement switches [34]). Such per-switch configurations require prior knowledge about routing and traffic distribution. Acquiring this information incurs substantial overheads in setting up the measurement infrastructure. Furthermore, network and traffic dynamics require continually updating these configurations, which further complicates the solution. Our method avoids such overheads and requires no information on routing, workload, or network topology. It is also the first to perform flow sampling without such information.

Supporting a wide range of measurements. AROMA supports numerous controller algorithms that utilize the packet and flow samples to accomplish a variety of network measurement tasks, such as estimating the number of (different) packets and flows in the measurement, estimating per-flow

frequency, identifying the heavy hitter flows, calculating hierarchical heavy hitters, estimating the flow size distribution, and identifying super-spreaders.

PISA compatible. AROMA can be implemented using P4 and deployed on PISA (Protocol Independent Switch Architecture) programmable switches. Furthermore, the overall switch resources required by AROMA are minimal, which allows the switch to perform additional functionality, giving AROMA an distinct advantage over existing monitoring techniques [7], [28], [29], [36].

Accurate network-wide measurements. Evaluation on real-world network traces shows AROMA provides accurate measurements for a variety of network tasks. Additionally, it is close in performance to existing solutions that provide similar guarantees, but cannot be implemented using the limited resources and functionality of PISA switches. For example, with just 0.5MB of space on a trace of 32 million packets: AROMA estimates flow sizes with a root mean square error of just 150 packets, achieves an F1 score of 0.9 in identifying superspreaders and of 0.8 in finding heavy hitters, and estimates the flow size distribution with a weighted mean relative difference of just 0.045.

II. The AROMA Framework

In a nutshell, AROMA collects flow and packet samples within the data plane of programmable switches. AROMA guarantees that all packets (or flows) have an equal chance to be sampled. As a result, our system is routing oblivious and its output is mathematically identical regardless of network topology and routing.

The controller merges the samples to form a uniform network-wide sample. Finally, the controller uses the samples to estimates various statistical properties. AROMA is partitioned into a data plane module that stores and maintains the samples, and a measurement analysis module which runs on the controller. In addition, AROMA leverages a combination of packet sampling and flow sampling (as in [34]), which together allow our system to be general and support a large variety of tasks [35].

Specifically, AROMA uses a two phase hash-based sampling technique to first select a sample slot, and each slot retains the element with minimal hash value. Here, an element is either a packet (in packet sampling) or a flow (in flow sampling). The controller then merges the sampling slots from all switches, and attains for each slot the element whose hash value is (globally) minimal. That is, an element is sampled only if its hash value is globally minimal for its corresponding slot. Finally, AROMA runs on P4 programmable switches, and naturally fits within the switch constraints. Furthermore, our solution requires a minimal number (2-3) of pipeline stages and can be configured to run with any amount of memory. This allows AROMA to operate alongside higher-level applications such as load balancing or attack detection.

We first formally define our model, assumptions, and the notations (Section II-A). Next, we show a method to store and collect a uniform sample (of flows, or packets) within the data plane using PISA programmable switches (Section II-B). Subsequently, we present the controller algorithm to merge the samples collected distributively into a network-wide uniform sample, and survey ways to utilize the samples to perform various measurement tasks (Section II-C).

Symbol	Definition
$\mathcal{S}$	The packet stream
$\langle fid_i, pid_i \rangle$	A packet from flow fid_i and packet identifier pid_i
$\mathcal{U}$	The universe of flow identifiers
f_x	The frequency of flow $x \in \mathcal{U}$
$\widehat{V}$	an estimate for $\lvert\mathcal{S}\rvert$
ε	The goal error parameter
δ	The goal error probability
M	The number of samples required for the accuracy guarantee
α	A space factor that allows faster convergence ($\alpha \geq 1$). We use $\alpha \cdot M$ slots instead of M.
$\widetilde{M}$	The number of samples the algorithm produced ($\widetilde{M} \in [0, \alpha \cdot M]$)
$\widehat{p}$	The estimated sampling probability ($\widehat{p} = \widetilde{M}/\widehat{V}$)
T	The actual sample produced ($\lvert T\rvert = \widetilde{M}$)
T_x	The number of times x appears in the sample T
$\widehat{f_x}$	An estimate for the frequency of flow x (i.e., $\widehat{f_x} = T_x/\widehat{p}$)
θ	Heavy hitter threshold
$\widehat{M}(t)$	The number of samples the algorithm produced by time t

TABLE I: List of symbols and notations.

A. Preliminaries

We model the traffic as an ordered *stream* of packets $\mathcal{S} \in (\mathcal{U} \times \mathbb{N})^*$, where each packet $\langle fid_i, pid_i \rangle$ has a unique *flow identifier* $fid_i \in \mathcal{U}$, and a *packet identifier* $pid_i \in \mathbb{N}$ that matches each packet to a single flow. Flow identifiers can be source IPs, source and destination IP pairs, or 5-tuples. For packet identifiers, in the case of TCP, we can use the TCP sequence number as part of a unique packet identifier. In general, the works of [16], [43] explain how the packet header fields can be used to derive unique packet identifiers. In this work, we assume the existence of unique packet identifiers.

We use K *measurement switches* $R_1, \ldots R_K$, and assume that each packet traverses *at least one* measurement switch. Yet, some packets may traverse multiple measurement switches, and the routing rules may change during the measurement. Formally, our only assumption is that suppose each switch sees a subset of the stream ($\mathcal{S}_k \subseteq \mathcal{S}$), then all the switches together cover all the packets $\cup_{k=1}^{K} \mathcal{S}_k = \mathcal{S}$.

Our model is more general than that of other network-wide measurement solutions that assume that packets only visit a single measurement switch [24], [25], [39], or each flow is routed through a single fixed path [28]. The same model is also used in related work [3], [4] in a software context.

The term flow refers to the set of packets that share the same flow identifier. Given a flow identifier x, its *frequency* f_x is the number of packets with x as their flow identifier, i.e., $f_x = \lvert\{i | fid_i = x\}\rvert$.

B. Data Plane Sampling Module

We now introduce our data plane sampling infrastructure. Section II-B1 provides a high-level overview of the algorithm, while Section II-B2 provides the P4 implementation details, adhering to the PISA architecture.

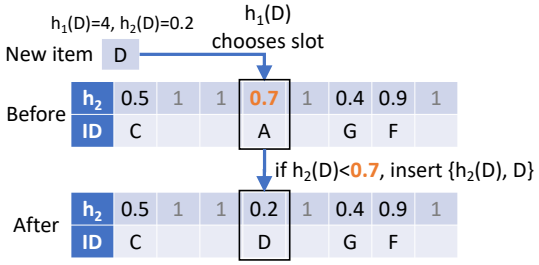


Fig. 1: We compute two hashes for each observed item (packet or flow). h_1 determines in which slot to compete; if the slot is empty we add the new item. Otherwise, we add it only if its h_2 value is smaller than that of the stored item.

1) Algorithm overview: Each measurement switch allocates a fixed size block of memory for $(\alpha \cdot M)$ slots. Measurement switches calculate a hash value in $(0, 1]$ based on the packet (or flow) identifier. Each slot stores the item (packet/flow) with the minimal hash value from all the items that were assigned to the slot. Interestingly, hash collisions mean that we require time before the slots are filled. We show that the number of filled slots behaves like a variant of the Coupon Collector problem, but instead of trying to fill **all** the slots like the common analysis, we attempt to fill a certain percentage of all slots (say 90%). This relaxation asymptotically reduces the time required to collect the sample at the expense of a slightly inflated memory consumption.

Formally, each measurement switch observes a stream of packets $\langle fid_i, pid_i \rangle$, with a flow identifier fid_i and packet identifier pid_i. We denote x_i as the identifier used, i.e., $x_i = fid_i$ for flow sampling and $x_i = pid_i$ for packet sampling.

Each switch maintains a data structure MEM, which contains $(\alpha \cdot M)$ memory slots, and each slot stores exactly one identifier. The value $\alpha \geq 1$ is selected to ensure a sample of size at least M. We provide an analysis of the convergence time in Section III. Let us denote memory slot j as $MEM[j]$. Two values are maintained within the slot: a hash value $MEM[j].hash$ and an identifier $MEM[j].id$.

We use two independent random hash functions: $h_1 : \mathcal{U} \rightarrow [0, \alpha \cdot M)$ for mapping an identifier to a memory slot, and $h_2 : \mathcal{U} \rightarrow (0, 1]$ to decide which item to sample. Imperatively, the hash functions are *identical* for the measurement switches that participate in the measurement as it allows merging the data structures for obtaining a network-wide uniform sample and thereby a global view. At a high level, each slot receives a fraction of incoming packets, and stores a single identifier x that has the smallest $h_2(x)$ of all those observed by that memory slot. We assume that all measurement switches use *the same* hash functions and that these are chosen at random. This can be done by having the controller randomly choose a seed for pseudo random generator uniformly for all switches at the beginning of the computation epoch.

We initialize all $MEM[j].hash$ to 1. As illustrated in Figure 1, upon observing each packet and determining identifier $x_i =$ "D", the switch does the following:

Algorithm 1: Maintaining uniform flow samples

```
1  control FlowSampling(  inout headers hdr,
                          inout metadata meta) {
2      register< bit<32> > (1≪m) MEM_hash;
3      register< bit<64> > (1≪m) MEM_id;
4      apply {
           //Prepare flow identifier x
5          meta.flowid[31: 0]=hdr.ipv4.srcAddr;
6          meta.flowid[63:32]=hdr.ipv4.dstAddr;
           //Compute hash h1(x) ∈ [0, 2^m),
             h2(x) ∈ [0, 2^32)
7          hash(meta.h1, HashAlgorithm.crc32, 0, {meta.flowid},
             1≪m);
8          hash(meta.h2, HashAlgorithm.crc32_custom, 0,
             {meta.flowid}, 1≪32);
9          bit<32> existing_sample_h2;
10         MEM_hash.read(existing_sample_h2, meta.h1);
           //If the current packet has smaller
             hash, replace existing sample
11         if (meta.h2<existing_sample_h2){
12             MEM_hash.write(meta.h1, meta.h2);
13             MEM_id.write(meta.h1, meta.flowid);
14         }
15     }
16 }
```

1) Computes the two hash values $h_1(x_i)$ and $h_2(x_i)$ for the current packet. In the example above, $h_1(x_i)$ is the fourth column, and $h_2(x_i) = 0.2$.
2) Looks up the hash value stored in $MEM[h_1(x_i)].hash$, and ignores the packet if $MEM[h_1(x_i)].hash \leq h_2(x_i)$.
3) Otherwise, if $h_2(x_i) < MEM[h_1(x_i)].hash$, then we replace the existing sample in the slot:

$$\begin{aligned} MEM[h_1(x_i)].hash &\leftarrow h_2(x_i) \\ MEM[h_1(x_i)].id &\leftarrow x_i \end{aligned}$$

We want to run two instances of our sampling algorithm simultaneously, one for packet-sampling and the other for flow-sampling. Recall that we select $x_i = pid_i$ to sample packets, and $x_i = fid_i$ to sample flows.

For correctness and accuracy guarantees, we require that at least M out of the $\alpha \cdot M$ slots will not be empty to obtain an M-sized uniform sample. We can choose $\alpha \geq 1$ to expedite this process; in practice, a choice of $\alpha = 1.5 \sim 2$ suffices.

2) Implementation on PISA programmable switches: To achieve Tbps-level aggregated throughput and low forwarding latency, a PISA [11] programmable switch uses a packet processing pipeline architecture that allows only simple operations per pipeline stage, and only has a certain number of hardware stages. The work of [7] summarized the limitations imposed by PISA switches. Most relevant to our case are the limited number of programmable pipeline stages, the limitations on memory access, and the limitations on arithmetic operations. AROMA's P4 implementation requires $O(1)$ memory accesses per packet, and can be implemented using only 2 pipeline stages. Thus, it leaves plenty of room for the measurement switch to run other network applications.

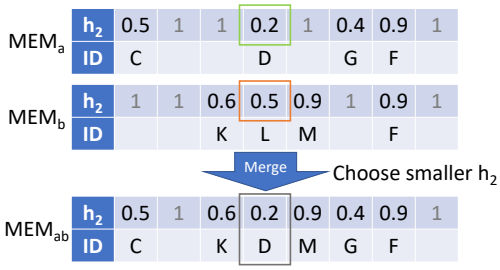


Fig. 2: An example of the merge process of two samples collected in different measurement switches. When the same slot contains different items, we select the one whose h_2 value is smaller. In this example, we select D and discard L.

We now discuss some of the implementation details for performing uniform sampling in the data plane. At each programmable switch, we allocate two register memory arrays each with $\alpha \cdot M$ entries. Note that we select α such that $\alpha \cdot M = 2^m$ for some $m \in \mathbb{N}$. Such a selection simplifies the implementation as we only have access to random bits, and thus randomizing a number in an arbitrary range is more difficult to implement, and incurs additional overheads.

We denote the register arrays as $MEM^{hash}[i]$ and $MEM^{id}[i]$, and store them in adjacent pipeline stages. The power-of-two sizing of the arrays allows easy addressing using an m-bit hash function h_1.

Since each array entry on hardware switches usually stores 32 bits, we store $h_2(x) \in (0, 1]$ into MEM^{hash} using 32-bit fixed point encoding. Hash functions h_1 and h_2 are implemented using CRC32 with different polynomials, and h_1 is truncated to m bits. We also initialize all entries in MEM^{hash} to 1.

As demonstrated by the P4 code shown in Algorithm 1, for each incoming packet $\langle fid_i, pid_i \rangle$, the programmable switch determines x_i ($x_i \in \{fid_i, pid_i\}$) and does the following:

1) Access parsed header fields, such as IPv4 source and destination addresses, to retrieve x_i (line 5,6), then compute $h_1(x_i) \in [0, 2^m)$ and $h_2(x_i) \in (0, 1]$ (line 7,8).
2) Compare the value found in $MEM^{hash}[h_1(x_i)]$ to $h_2(x_i)$.
   - If $MEM^{hash}[h_1(x_i)] <= h_2(x_i)$: do nothing.
   - If $MEM^{hash}[h_1(x_i)] > h_2(x_i)$, replace the existing entry with the following as shown in line 12 and 13:
     $MEM^{hash}[h_1(x_i)] \leftarrow h_2(x_i)$,
     $MEM^{id}[h_1(x_i)] \leftarrow x_i$,

The algorithm uses minimal resources; it computes two hash functions, and resides within two hardware pipeline stages. Thus, we can simultaneously implement both flow-sampling and packet-sampling in existing programmable switch targets.

C. Using the Samples in Control Plane

The controller merges samples collected from all switches to form a global uniform sample set as described in Section II-C1. In the subsequent sections, we briefly describe how the controller uses the sample set to perform various measurement tasks.

1) Merging samples: First, we describe how to merge two samples into a single sample, as illustrated in Figure 2. Repeatedly applying this algorithm allows the controller to merge all the samples.

Given the samples collected by two switches, $MEM_a[\cdot]$ and $MEM_b[\cdot]$, AROMA merges them to $MEM_{a\cup b}[\cdot]$ as follows: it iterates over the $\alpha \cdot M$ slots, and for each slot j it compares $MEM_a[j].hash$ and $MEM_b[j].hash$ to select the smaller of the two values as the new value for $MEM_{a\cup b}[j].hash$. It then sets $MEM_{a\cup b}[j].id$ accordingly.

It is straightforward to prove that the resulting values in $MEM_{a\cup b}$ are the same as if all packets were observed by either switch a or b or both, as the smaller h_2 hash value in each slot will prevail. We repeat this process to merge the samples collected at all the switches, to obtain the global sample $MEM_{global}[j].\{hash, id\}$. We further trim $MEM_{global}[\cdot]$ to ignore empty slots.

2) Number of packets/flows: Perhaps the most fundamental measurement task is to estimate the actual number of packets and flows within the measurement. In the routing oblivious setting, this is not equivalent to summing the number of flows or packets over the different measurement switches as we do not know how many measurement switches each of them traversed. In the HyperLogLog algorithm [19], the stream is partitioned into separate substreams, and an independent estimator is maintained for each substream. Each estimator maintains the longest run of consecutive leading zeros of the randomly hashed output of items in its substream. The estimator than uses this information to determine the number of distinct items in each substream. Similarly to HyperLogLog, by looking at the hash value in each slot, we can infer how many distinct identifiers contested for that slot.

More precisely, given the hash value stored in the i'th slot is $MEM_{global}[i].hash \in (0, 1]$, the expected total number of distinct identifiers hashed to this slot is $\frac{1}{MEM_{global}[i].hash}$. Thus, by scaling up the harmonic mean of each slot's estimate, the total number of distinct identifiers seen by all the slots can be estimated by: $\widehat{V} = \frac{(\alpha \cdot M)^2}{\sum_{i=0}^{\alpha \cdot M - 1} (MEM_{global}[i].hash)}$.

Given this estimated number of different packets/flows, we also get an estimation of the sampling probability. For that, assume that $\widetilde{M} \in [0, \alpha \cdot M]$ slots were filled (i.e., we have a uniform sample of size $\widetilde{M}$); then the *estimated* sampling probability is $\widehat{p} = \widetilde{M}/\widehat{V}$. We require the estimated sampling probability for other measurement tasks (such as frequency estimation, superspreaders, and frequency distribution estimation).

3) Distributed frequency estimation: To estimate flow size f_x for a flow x, we inspect the uniform packet sample set and look at the packet identifiers. We denote by $T_x = |\{0 \le i \le \alpha \cdot M \mid MEM_{global}[i].id \in x\}|$ the number of packets in the global uniform sample set that belong to flow x. Subsequently, we divide T_x by the estimated sampling probability $\widehat{p}$ to get estimated flow size $\widehat{f_x} = T_x/\widehat{p}$.

4) Distributed heavy hitters: We can use the uniformly sampled packets to estimate heavy hitters, defined as those flows with size f_x which exceeds a θ fraction of total packet

traffic ($|\mathcal{S}|$), i.e., $f_x > \theta \cdot |\mathcal{S}|$. Our algorithm outputs every flow whose frequency in the sample is at least a θ-fraction of the sample size. For example, if $\theta = 1\%$ and we gathered $\widetilde{M} = 10000$ samples, we will output every flow that appears in the sample at least 100 times. If an application is more recall-oriented or precision-oriented it is possible to change the threshold to get (with high probability) 100% accuracy in one of them (at the cost of degrading the other).

5) Hierarchical heavy hitters: We look at the uniformly sampled packets to determine hierarchical heavy hitters. We report a prefix as a hierarchical heavy hitter if it appears in more than $\theta \cdot \widetilde{M}$ packets.

6) Superspreaders: We define a Superspreader as a source IP address that communicates with more than Ψ destination IP addresses. Such an IP address appear in many flows and is therefore likely to appear in the uniform flow sample. Given a uniform sample of $\widetilde{M}$ flows, we can examine the flow identifiers and see if any source IP address appeared more than $\Psi \cdot \widehat{p}$ times; such a source IP is sending out to more than Ψ destination IPs in expectation.

III. Analysis

In this section, we provide rigorous bounds on the accuracy of the algorithm. As a general note, we refer here to a *packet stream* which can be distributed in any way between the measurement switches as long as each packet is measured at least once. All the results in this section are also applicable to flow sampling by replacing the notion to "flow stream". Specifically, we analyze the guarantee for estimating flow-sizes which, by simple reductions, also extend to Heavy Hitters (HH) and Hierarchical HH (HHH). For superspreaders the analysis is also applicable, although the condition regarding the minimum number of packets (M) is replaced by similar lower bound on the number of flows. The entire section assumes that the hash functions are independent and are $\Omega(M)$-wise independent. In practice, simpler hash functions often suffice [12].

Our goal is to estimate flow sizes, with high probability, up to an additive error of $|\mathcal{S}| \cdot \varepsilon$. This type of guarantee is standard in streaming algorithms and appears in [4], [30], [13], [38] and many others. However, due to the nature of our algorithm, we cannot provide this guarantee immediately but rather require *convergence time*. Recall that we first apply h_1 and map the packet into a slot that holds a single sample. Thus, it may take a while to achieve a large enough sample as multiple packets may be hashed to already full slots. Formally, *hash collisions* in h_1 mean that some packets may not be sampled even if not all slots are full.

We mark by $\widetilde{M}(t) \in [0, \alpha \cdot M]$ the number of non-empty slots in our algorithm after seeing t packets. We utilize the result of [4] that shows the accuracy guarantee one gets from analyzing a uniform sample of size M. By symmetry, we have that if $M(t)$ slots are filled, any subset of $M(t)$ packets has the same probability of appearing in the sample and therefore the sampling is uniform. We say that our algorithm has *converged* once $\widetilde{M}(t) \geq M$ and thus we provide the accuracy guarantee.

Lemma 1. *([4]) Let $T \subseteq \mathcal{S}$ be a random packet subset of size $M \geq \left\lceil 3\varepsilon^{-2} \log_2(2/\delta) \right\rceil$. For a flow $x \in \mathcal{U}$, let T_x be its frequency in T. Then* $\Pr\left[\left| f_x - T_x \cdot |\mathcal{S}|/M \right| \geq |\mathcal{S}|\varepsilon\right] \leq \delta$.

If $\alpha = 1$, then the process of collecting the samples from the nonempty slots is known as the *Coupon Collector* problem [10]. In the Coupon Collector problem, a collector wishes to gather all M coupons while getting a single coupon, uniformly at random, at each step. Since the time to collect the i'th distinct coupon is distributed geometrically with mean $M/(M-i)$, we have that the expected time to collect all coupons is $\sum_{i=1}^{M} M/(M-i) = M \ln M + O(M)$. To derive a high-probability bound, observe that the probability that a given coupon is not collected after r steps is $(1-1/M)^r \leq e^{-r/M}$. By using the union bound and setting $r = M \ln(M/\delta))$ we get that $\Pr[M(r) < M] \leq M \cdot e^{-r/M} \leq \delta$. This analysis is directly applicable to our method for $\alpha = 1$ as we uniformly hash every packet into one of the M slots and the goal is to fill all slots. We can then choose M to guarantee the desired result with probability $1 - \delta/2$ and use the union bound to derive the standard (ε, δ)-guarantee. We summarize this in the following theorem.

Theorem 1. *For any $\varepsilon, \delta > 0$, let $M = \left\lceil 3\varepsilon^{-2} \log_2(4/\delta) \right\rceil$; our algorithm (with $\alpha = 1$ and thus M slots) guarantees approximating flow sizes up to an $(|\mathcal{S}|\varepsilon)$-additive error, with probability $1-\delta$, given that the number of packets it processes is at least $M \cdot \ln(2M/\delta)$.*

The above solution works well if the measurement is long enough with respect to the error parameters ε and δ. However, this may prove to be too lengthy for accurate measurements. For example if $\varepsilon = \delta = 1\%$ we guarantee the convergence of the algorithm after about 4.4 million packets.

To shorten the convergence time, we explore the *space to convergence time* tradeoff that α values larger than 1 offer. Schematically, by increasing α we pay a constant factor in the amount of space required, but reduce the convergence time asymptotically, as we now show. We note that while the coupon collector analysis above is standard, to the best of our knowledge, the process described in this section is novel to our work. Particularly, we get that for any constant $\alpha > 1$ the number of packets until convergence drops to $O(M)$, as summarized in the following theorem.

Theorem 2. *Let $M \in \mathbb{N}^+, \alpha > 1$ and denote $\beta = 1 + 1/\ln\alpha + \ln(2/\delta)/(M \cdot \ln\alpha)$. When allocated with $\alpha \cdot M$ slots the algorithm fills at least M of them after seeing $\beta \cdot M$ packets with probability at least $1 - \delta/2$.*

Proof. For a subset of indices $K \subseteq [\alpha \cdot M]$ of size $|K| = M - 1$, define by I_K an indicator for the event that all packets were mapped only into the indices of K. This event is bad as it means that the algorithm cannot produce an M-sized uniform packet sample and fails to provide the approximation guarantee. Observe that the probability of this event is $\Pr[I_K] = \left(\frac{M-1}{\alpha \cdot M}\right)^{\beta \cdot M}$. Since the number of such

subsets K is $\binom{\alpha \cdot M}{M-1}$, we can use the union bound to get that the probability that such a subset exists is at most $\Pr\left[\exists K \subseteq [\alpha \cdot M] : |K| = M-1 \wedge I_K\right]$ $\leq \binom{\alpha \cdot M}{M-1} \cdot \left(\frac{M-1}{\alpha \cdot M}\right)^{\beta \cdot M} \leq (e \cdot \alpha)^M \cdot \left(\frac{1}{\alpha}\right)^{\beta \cdot M} = \delta/2$, where the last inequality follows from the known binomial coefficient bound $\binom{n}{k} \leq \left(\frac{e \cdot n}{k}\right)^k$. □

In the following theorem, we once again use Lemma 1 for providing the error guarantee. To exemplify the reduction in convergence time, consider the above parameters ($\varepsilon = \delta = 1\%$ and $\alpha = 2$ which means double space used). Our result implies guaranteed convergence after 630K packets, a reduction of over 85%.

Theorem 3. *For any $\varepsilon, \delta > 0$, let $M = \lceil 3\varepsilon^{-2} \log_2(4/\delta) \rceil$; AROMA (with $\alpha > 1$ and $\alpha \cdot M$ slots) approximates flow sizes within an $(|\mathcal{S}|\varepsilon)$-additive error, with probability $1-\delta$, after at least $M \cdot (1 + 1/\ln \alpha + \ln(2/\delta)/(M \cdot \ln \alpha))$ packets.*

IV. Evaluation

Dataset: We used the CAIDA Anonymized Internet Trace 2018 [1]. The trace contains internet packets collected from the "equinix-nyc" high-speed monitor. For each packet, we use its 5-tuple (anonymized source-destination IP pair, port pair, and protocol) as its flow ID. We summarize the number of distinct flows, for a given stream length, in Table II.

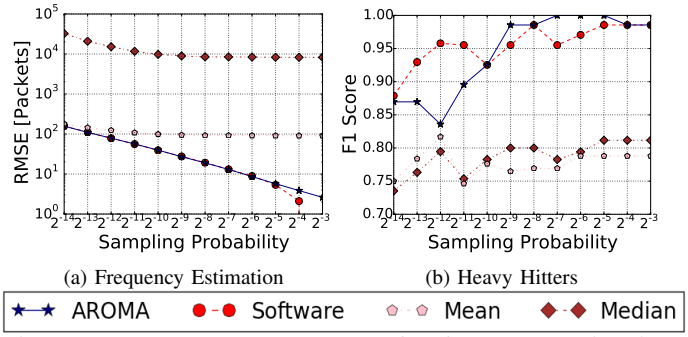
Length	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
#flows	15K	26K	41K	66K	107K	183K	314K	550K	967K	1.69M

TABLE II: The number of distinct 5-tuples in the measurement as a function of the number of packets in the trace.

Metrics: We consider the following performance metrics:

1) Root Mean Square Error (RMSE): Measures the differences between predicted values of an estimator to actual values. Formally, for each flow x the estimated frequency is $\widehat{f_x}$ and real frequency is f_x. RMSE is calculated as: $\sqrt{\frac{1}{|\mathcal{U}|}\sum_{x \in \mathcal{U}}(\widehat{f_x} - f_x)^2}$.
2) F1 Score: A quantity that combines precision (the correct fraction of reported flows), and recall (the fraction of true flows that were reported) into a single numerical value in the following manner: $F1 = 2 \cdot (precision \cdot recall)/(precision + recall)$.
3) Weighted Mean Relative Difference (WMRD): consider the set of flow sizes $\{f_x | x \in \mathcal{U}\}$ and let z be the size of the largest flow. Denote by $F_i = |\{x \in \mathcal{U} | f_x = i\}|$ denote the number of flows of size i. Let $\widehat{F_i}$ be the estimation produced by an algorithm for F_i. Define the sum of absolute errors to be $E = \sum_{i=1}^{z} \left|F_i - \widehat{F_i}\right|$ and the sum of averages as $A = \sum_{i=1}^{z}(F_i + \widehat{F_i})/2$. The metric is then defined as $WMRD = E/A$. WMRD is always between 0 and 2 with a perfect match being 0 and complete disagreement being 2.

Evaluation Parameters: In Figures 3-5 we used the first $2^{25} \approx 33.55$ million packets. The x-axis in these plots is the allocated *per-switch* space. We define a heavy hitter as a flow whose size is at least 0.1% of the overall number of packets in the measurement. For the first 2^{25} packets of New York 2018 dataset this amounts to 35 heavy hitters. Similarly, we define a hierarchical heavy hitter as a source network that appear in more than 0.1% of the overall traffic; this follows the HHH definition of [41]. We define a superspreader as a source IP that communicated with at least $\Psi = 1000$ distinct destination IPs. For these parameters, we measured 54 such sources. In Figures 6-7, where the number of packets varies, we keep the 0.1% threshold for HH and HHH and set the SS threshold such that there are ≈ 50 superspreaders for each point.

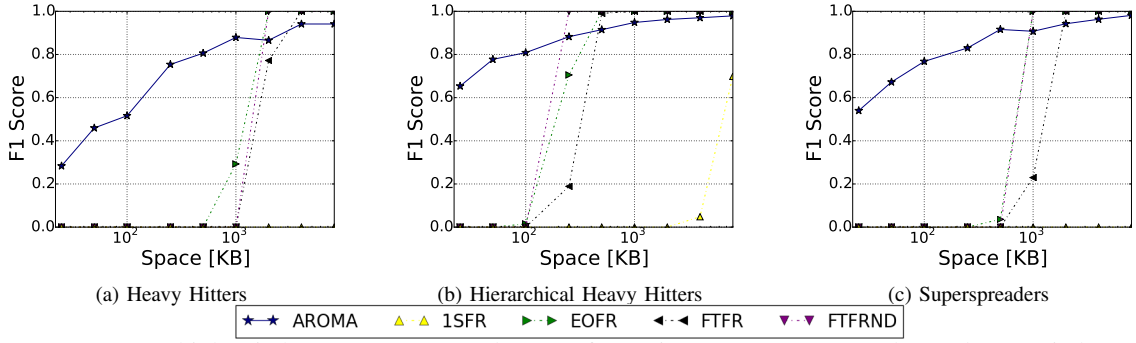


(a) Frequency Estimation (b) Heavy Hitters

Fig. 3: Root Mean Square Error for frequency estimation (lower is better), and F1 score for heavy hitters (higher is better), when comparing our method to (plain) random sampling, on an Internet-like hop count distribution.

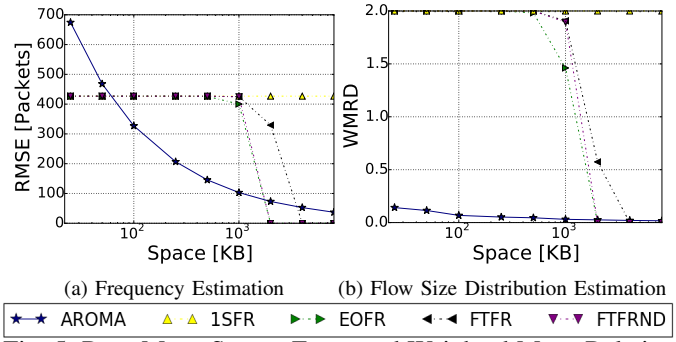
Comparison with uniform sampling and software solutions: We start comparing AROMA's accuracy to naive uniform sampling. Recall that in such sampling, packets get an opportunity to be sampled for each measurement switch they visit, which biases the controller's sample. Additionally, we compare AROMA to the BEFMR18 software routing oblivious algorithm of [4]. We used the Internet's hop count distribution by [18], [37], and assumed a measurement switch at each hop which improves reliability. Specifically, this model assumes that the probability for k-hops (for a given flow) is: $\Pr[k \text{ hops}] = \frac{1+o(1)}{N}\sum_{m=0}^{k} c_{m+1}\frac{(\ln N)^{k-m}}{(k-m)!}$, where c_i is the i'th Taylor coefficient of the reciprocal of the Gamma function $1/\Gamma(z)$ [2, Table 6.1.36]. The work of [37] models the actual hop-count distribution of the Internet as the distribution for $N = 98400$, which gives a median hop count of 12.

We deploy measurement switches on each hop and normalize the frequency at the controller by either the mean or the median hop count, of the hop count distribution. Figure 3 shows the results of this evaluation, where Median (Mean) is the uniform sampling normalized by the median (mean) value, AROMA is our algorithm and Software refers to [4]. Figure 3a shows the results for estimating per flow frequency. The mean normalization provides better accuracy for (plain) random sampling. Our method and Software are almost identical and are considerably more accurate for a wide range of sampling probabilities. Figure 3b shows the F1 score for heavy hitters; our method and the Software method offer higher F1 values than uniform sampling. Note that for this application, it is unclear which normalization (mean or median) is superior. Intuitively, uniform sampling suffers from flows whose hop-count significantly differs from the mean or median and thus



(a) Heavy Hitters (b) Hierarchical Heavy Hitters (c) Superspreaders

Fig. 4: F1 scores (higher is better) on New York 2018, for various measurement tasks, and per-switch space.



(a) Frequency Estimation (b) Flow Size Distribution Estimation

Fig. 5: Root Mean Square Error, and Weighted Mean Relative Difference (lower is better), for various tasks on the New York 2018 dataset.

are grossly underestimated or overestimated.

Comparison with existing hardware solutions: Figure 4 shows an evaluation of AROMA's performance when compared to existing works, for various measurement tasks. We also compared with FlowRadar [28] in various configurations. Throughout the evaluation, we consider FlowRadar's estimation of the size of a flow which it failed to decode as zero. For the FatTree topology we assume that the flows are distributed uniformly, the easiest setting for FlowRadar.

- **`1SFR`** - FlowRadar where all packets go through a single switch.
- **`FTFR`** - FlowRadar deployed on all switches of a k=8 fat tree with FlowDecode (the faster decoding procedure).
- **`FTFRND`** - FlowRadar deployed on all switches of a k=8 fat tree with NetDecode (the more accurate but slower decoding procedure).
- **`EOFR`** - FlowRadar deployed on all edge switches of a k=8 fat tree with packets only measured once (on the first edge switch they visit).

Figure 4a shows the F1 metric for heavy hitter measurement (higher F1 values are better). FlowRadar (in all scenarios) fails to provide any meaningful information until circa 1 MB of space, and from that point on, it rapidly improves with more space until it provides an exact measurement (F1=1) which is better than our approach. We conclude that FlowRadar has a stricter minimum memory size requirement while AROMA performs better under a tight memory budget, which is expected in today's switches, with a few MBs of data-plane memory shared among different measurement tasks.

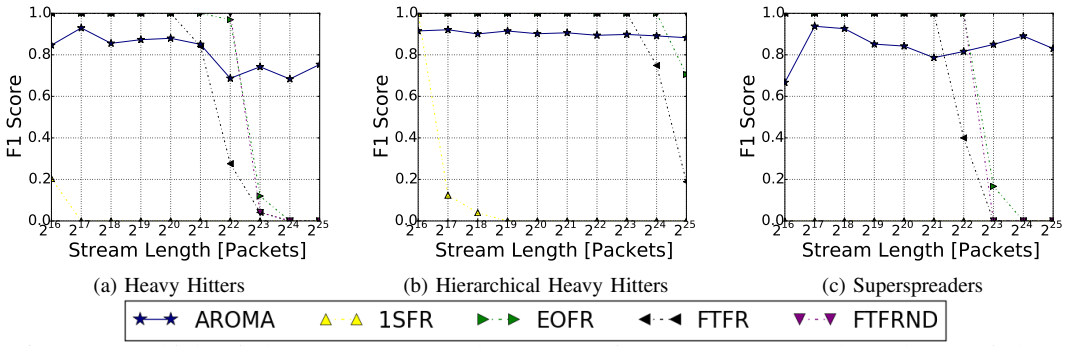
Figure 4b show results for the hierarchical heavy hitters' task, and Figure 4c for superspreader measurements, as can be observed the qualitative behavior is the same as in the heavy hitter case. AROMA can operate and provide accurate measurements while FlowRadar fails unless it is allocated with enough space. Thus, for these tasks, our algorithms are superior when given a small amount of space, and inferior when there is enough space to run FlowRadar efficiently. Recalling Table II, we observe that the actual number of flows increases with the measurement length. Thus, we expect our method to be more reliable in long measurements, especially as traffic anomalies such as port scan attacks can increase the number of flows in the measurement.

Figure 5 shows results for our packet sampling algorithm and the frequency estimation problem. As well as, for our flow sampling algorithm, and the flow size distribution estimation problem. In Figure 5a, we can see that our approach continuously improves given more space. In contrast, the various FlowRadar configurations are very inaccurate until there is enough memory, and then they have no error at all. Still, AROMA outperforms FlowRadar in many configurations.

In Figure 5b, we see results for the flow size distribution estimation task. In the FlowRadar configurations, we again see the "cliff" where the algorithms do not work until there is enough memory allocated. Notice that the required memory for them to work is several megabytes, whereas our algorithm is accurate even with a few kilobytes.

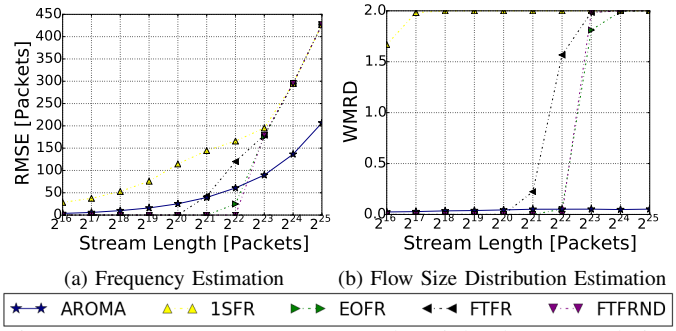
Next, we allocate 250KB for each switch and monitor the accuracy throughout the trace. Figure 6 shows the F1 score for different applications and varying the stream length. Initially, FlowRadar configurations achieve accurate measurement (F1 score of 1). Then, as the measurement prolongs, we encounter more flows, and FlowRadar configurations begin to fail. Once we reach 32 million packets, all the FlowRadar configurations become ineffective. In contrast, our sampling-based approach is relatively accurate throughout the measurement. We conclude that AROMA is superior when there is insufficient memory space for an accurate measurement. While accurate measurements are desired, it is unclear how much memory FlowRadar would need to succeed, whereas in AROMA we always yield a relatively accurate outcome.

Figure 7 shows results for the frequency estimation, and flow size distribution estimation tasks (lower is better). In



(a) Heavy Hitters (b) Hierarchical Heavy Hitters (c) Superspreaders

Fig. 6: F1 score (higher is better) on New York 2018, varying measurement tasks and per-switch space.



(a) Frequency Estimation (b) Flow Size Distribution Estimation

Fig. 7: Root Mean Square Error and Weighted Mean Relative Difference (lower is better) on the New York 2018 dataset, varying the measurement's length.

Figure 7a, we see that the accuracy of our method gracefully degrades throughout the measurement. FlowRadar degrades accuracy less gracefully as the measurement prolongs. When the measurement is long enough, our approach is more accurate than all FlowRadar configurations. Figure 7b shows the flow size distribution estimation accuracy throughout the measurement. The results are qualitatively similar, but our method does much better than FlowRadar configurations.

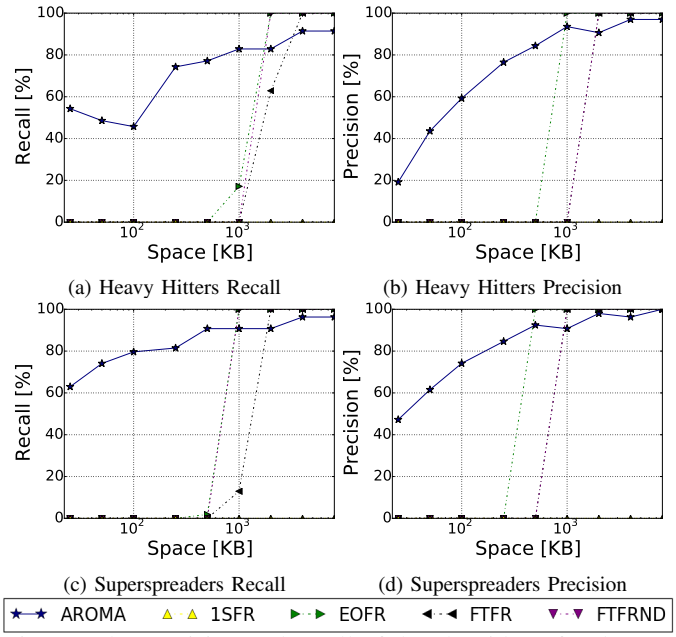
Performance breakdown: For the HH, HHH, and SS tasks, we used F1 as a single-metric for evaluating the performance of algorithms. However, the actual precision and recall performance of the algorithm is not the same. Our algorithms provide near-perfect precision and recall, while FlowRadar gives perfect precision but a poorer recall. The reason is that FlowRadar provides the exact sizes of the flows it decodes and thus know if one is a heavy hitter. We show the precision and recall performance in Figure 8.

V. Related work

We now survey network-wide techniques, as well as work that is related to the methods used in this work.

Packet Marking: The work of [3] suggests marking measured packets by exploiting unused bits in the IP header. That way, they measure each packet once regardless of the number of measurement switches it traverses. However, this simple and effective method implicitly restricts measurement switch deployment. Intuitively, the unused bits need to be cleared before they enter our network. Otherwise, the method may fail due to a proprietary use of these bits in other networks.

Single per-flow Path Solutions: FlowRadar [28], EverFlow, and Trajectory sampling [43], [16] assume that each flow



(a) Heavy Hitters Recall (b) Heavy Hitters Precision

(c) Superspreaders Recall (d) Superspreaders Precision

Fig. 8: The precision and recall of the algorithms for the HH and SS tasks. FlowRadar always has a perfect precision as any flow it decodes is fully accurate.

is routed on a single path. The single path solutions cannot handle routing changes, multicast, and Multipath routing [20] which are important in modern networks.

Flow Sampling Techniques: Several solutions have been proposed for flow sampling [34], [17]. Specifically, in [34], the authors present cSamp, a flow sampling method that performs hash-based packet selection to coordinate between the measurement switches. cSamp performs network-wide monitoring by distributing responsibilities across the measurement switches in the network. The framework is responsive to routing, topology and network dynamics and shifts the responsibilities according to the network changes. In contrast, AROMA achieves network-wide uniform flow sampling without assigning specific responsibilities and therefore is not affected by the network dynamics.

Other routing oblivious solutions: The BEFMR18 software-based algorithm [4] performs network-wide measurements through uniform packet sampling, using the same routing-oblivious assumption as this work. However, BEFMR18 is not compatible with the architecture of PISA

programmable switches, and it does not support flow sampling.

VI. Conclusion

We introduced AROMA, a network-wide measurement infrastructure that enables network-wide flow and packet sampling in PISA switches. AROMA does not make any assumptions regarding routing and is flexible with respect to the placement of the measurement switches in the network.

We proved formal accuracy guarantees and demonstrated the ability to perform a variety of network measurement tasks. We evaluated AROMA through simulations with different topologies, per-switch memory, and measurement length. AROMA outperforms uniform sampling and that it allows accurate measurements in memory-constrained configurations where the previous works are inapplicable.

AROMA's novelty extends beyond programmable switches. Specifically, it is the first technique to perform flow sampling without assumptions on the workload or coordination between the switches. Interestingly, it also has advantages in software implementation; specifically, it improves the update time of the existing (software) network-wide packet sampling technique [4] from logarithmic to a constant.

Acknowledgements: This work is supported in part by the CNS1834263 grant, the Ben-Gurion University Cyber-Security Research Center, and the Zuckerman Institute.

References

[1] The CAIDA equinix-nyc anonymized internet traces, 2018.

[2] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. Courier Corporation, 1965.

[3] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff. Detecting heavy flows in the SDN match and action model. *Computer Networks*, 2018.

[4] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz. Network-wide routing-oblivious heavy hitters. In *IEEE/ACM ANCS*, 2018.

[5] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. In *ACM SIGCOMM*, 2017.

[6] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard. Memento: Making sliding windows efficient for heavy hitters. In *ACM CoNEXT*, 2018.

[7] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE ICNP*, 2018.

[8] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM*, 2017.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, 2011.

[10] G. Blom, L. Holst, and D. Sandell. *Problems and Snapshots from the World of Probability*. Springer Science & Business Media, 2012.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, 2013.

[12] K. Chung, M. Mitzenmacher, and S. P. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 2013.

[13] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *ESA*, 2002.

[14] V. Demianiuk, S. Gorinsky, S. Nikolenko, and K. Kogan. Robust distributed monitoring of traffic flows. In *IEEE ICNP*, 2019.

[15] G. Dittmann and A. Herkersdorf. Network processor load balancing for high-speed links. In *Proc. of the 2002 Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems*, volume 735.

[16] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 2001.

[17] N. G. Duffield, C. Lund, and M. Thorup. Flow sampling under hard resource constraints. In *ACM SIGMETRICS*, 2004.

[18] A. Fei, G. Pei, R. Liu, and L. Zhang. Measurements on delay and hop-count of the internet. In *in IEEE GLOBECOM - Internet Mini-Conference*, 1998.

[19] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[20] A. Ford, C. Raiciu, M. J. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, 2013.

[21] P. García-Teodoro, J. E. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 2009.

[22] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[23] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[24] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM SOSR*, 2018.

[25] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.

[26] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, 2004.

[27] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *IEEE HOTI*, 2010.

[28] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *USENIX NSDI*, 2016.

[29] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[30] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *IN ICDT*, 2005.

[31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[32] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *Network, IEEE*, 1994.

[33] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*, 2015.

[34] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSamp: A system folr network-wide flow monitoring. In *USENIX NSDI*, 2008.

[35] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *ACM IMC*, 2010.

[36] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, 2017.

[37] P. Van Mieghem, G. Hooghiemstra, and R. Hofstad. A scaling law for the hopcount in internet. In *PAM*, 2001.

[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 *ACM SIGCOMM*, 2018.

[39] K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 2013.

[40] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *USENIX NSDI*, 2013.

[41] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *ACM IMC*, 2004.

[42] H. Zhao, A. Lall, M. Ogihara, and J. Xu. Global iceberg detection over distributed data streams. In *IEEE ICDE*, 2010.

[43] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.