Computer Networks 54 (2010) 1375–1388

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

An online framework for catching top spreaders and scanners

Xingang Shi [a,*], Dah-Ming Chiu [a], John C.S. Lui [b]

[a] Department of Information Engineering, The Chinese University of HongKong, Hong Kong
[b] Department of Computer Science and Engineering, The Chinese University of HongKong, Hong Kong

ARTICLE INFO

Article history:
Received 10 August 2009
Received in revised form 20 October 2009
Accepted 4 December 2009
Available online 16 December 2009
Responsible Editor: J. Neuman de Souza

Keywords:
Network monitoring
Sampling
Streaming
Small flow

ABSTRACT

Flow level information is important for many applications in network measurement and analysis. In this work, we tackle the "Top Spreaders" and "Top Scanners" problems, where hosts that are spreading the largest numbers of flows, especially small flows, must be efficiently and accurately identified. The identification of these top users can be very helpful in network management, traffic engineering, application behavior analysis, and anomaly detection.

We propose novel streaming algorithms and a "Filter-Tracker-Digester" framework to catch the top spreaders and scanners online. Our framework combines sampling and streaming algorithms, as well as deterministic and randomized algorithms, in such a way that they can effectively help each other to improve accuracy while reducing memory usage and processing time. To our knowledge, we are the first to tackle the "Top Scanners" problem in a streaming way. We address several challenges, namely: traffic scale, skewness, speed, memory usage, and result accuracy. The performance bounds of our algorithms are derived analytically, and are also evaluated by both real and synthetic traces, where we show our algorithm can achieve accuracy and speed of at least an order of magnitude higher than existing approaches.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

In network traffic measurement and analysis, flows are often used to represent communications between different end points. Fig. 1 shows a typical scenario in our network: hosts in a local ISP network communicate with other hosts in the global Internet through a high speed link, and we can monitor the traffic on that link. In this case, $host_3$ is communicating with a great number of other hosts, thus *spreading* a large number of flows. In many applications, such as detecting port or address scan, DDoS attack, worm propagation, and hot spots, the first step is to quickly and efficiently identify such kind of flow "spreaders". Yet, among those spreaders, what is the difference between a popular server and a malicious host scanning the network?

A malicious host scanning a lot of ports or addresses typically sends only a small number of probing packets to each victim to keep the overhead small and lower the chances to be detected, while a web server lets its users download content thus each flow may contain lots of packets. So flow size (in terms of the number of packets) can be a differentiator for spreaders. We refer to spreaders with many small flows as "scanners".

Let us consider some real-world examples. In a SYN flood attack, each flow typically contains only one SYN packet, discarding replied ACK packets. A low-rate TCP attack [23] uses many long-lived flows to exhaust resources of the victim, while the packet rates of these flows are kept low. Clients of online-games [2] may spread highly bursty and short UDP flows during server discovery. A bot server may send short commands to many bots under its control to initial an attack. P2P nodes search new peers and exchange control messages in a periodical fashion, and such messages typically contain a small number of packets

* Corresponding author. Tel.: +852 31634296.
E-mail addresses: sxg007@ie.cuhk.edu.hk (X. Shi), dmchiu@ie.cuhk.edu.hk (D.-M. Chiu), cslui@cse.cuhk.edu.hk (J.C.S. Lui).

1389-1286/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2009.12.003

(even fewer if the contacts fail). On the other hand, a popular server may have many long downloading sessions, and a P2P node usually keeps a few *good* connections for downloading, where flows typically have high rate and lots of packets. Generally speaking, the reason for monitoring large (elephant) flows is usually self-evident (e.g., for accounting and billing), and there is plenty of prior work [15]. The reasons for monitoring small (mice) flows are mostly due to security problems, and the monitoring of small flows is our focus in this work. Although intuitively it is easy to find hosts spreading many small flows by monitoring each flow and each host, the task is much harder on high speed links of a large network.

The rest of the paper is organized as follows: In the remainder of this section, we explain the problem formulation in more detail, and state our contributions. In Section 2, we review related work in this field. After that, we elaborate on two families of streaming algorithms for catching top spreaders and scanners in Sections 3 and 4, and combine them into a three-level framework in Section 5. We describe our evaluation methodology and results in Section 6, and conclude in Section 7.

1.1. Problem formulation

We formulate the "top spreaders" and "top scanners" problems as follows:

(1) A *user* is defined as a host and the user ID is just the IP address of that host. In the rest of this paper, the terms user and host will be used interchangeably.
(2) A *flow* is defined as a group of packets sharing common value for some fields, e.g., IP address, port, TCP flags, or even packet content. Here we use the commonly used 5-tuple $f = \langle \text{srcIP}, \text{dstIP}, \text{srcPort}, \text{dstPort}, \text{proto} \rangle$. The lifetime of a flow is not explicitly considered, since (i) a measurement interval naturally bounds the lifetime, and (ii) during a short interval, a flow is unlikely to terminate and start again.
(3) We consider small flows that *have no more than q packets*, which are closely related to security problems.
(4) The *top-k spreaders*[1] are users who are ranked within top k by their total flow counts and have at least T flows. Similarly, the *top-k scanners* are users who are ranked within top k by their numbers of small flows (flows of size no greater than q) and have at least T small flows. Using ranking alone may find users with only a small number of flows, while using threshold alone may suffer from the difficulty of choosing an appropriate threshold. Our definition combines these two constraints together and the exact value of T is less important. Besides, although theoretically k can vary from one to the total number of users, we usually consider a small k, e.g., 20 or 30, which is most useful for network management and security. By our online framework, we can answer questions like "What are the top 20 hosts that are spreading the largest number of small flows of no more than three packets?" at the end of a measurement epoch.

Our objective is to study online algorithms for catching those top users (spreaders or scanners) and accurately estimating their counts of specific flows (total flows or small flows). To keep up with the growth of the network, the algorithms must be scalable with respect to: *traffic scale*, that is the number of users and flows; *traffic skewness*, that is the per user flow count distribution; *traffic speed*, that is the packet arriving rate; *result accuracy*, which includes identified top users and their flow counts. In a nowaday ISP network, millions of users may send millions of flows during a small interval, and the wire-speed may be up to 40 Gbps leaving only tens of nanoseconds for processing each packet arrival. With such a stringent time constraint, only a very small amount of memory should be used so that it can reside in high speed SRAM to keep up with the link speed.

1.2. Our contribution

Our main contribution can be summarized as follows:

(1) We propose two streaming algorithms with provable performance bounds for counting small flows, especially singleton flows (flows with only one packet), of top scanners. To our knowledge, we are the first to address this problem in a streaming way. These algorithm can also be used for counting flows of top spreaders without any trouble.
(2) We propose a novel three-level framework which combines the above two algorithms and can catch top spreaders or scanners online. The first level, *filter*, translates arriving packets into a certain type of flow information; the second level, *tracker*, dynamically tracks the potential top users; and the third level, *digester*, maintains accurate flow information digest for these potential top users. In this framework, the two algorithms effectively help each other to improve accuracy while reducing memory usage and processing time, and work much better than individually used.
(3) We demonstrate through extensive experiments that our framework can get at least an order of magnitude better result for the "top spreaders" problem than existing approaches, and several orders of magnitude better for the "top scanners" problem than the straightforward application of sampling methods, in both speed and accuracy.

2. Related work

In this section, we review existing works related to our problem. Since catching top users requires knowledge of their flow counts and rankings, we begin with some background on these basic problems.

[1] This "top spreaders" problem has been studied in some earlier work [33,19] without considering the threshold T.

2.1. Sampling and streaming

To deal with a huge number of packets, flows and users in real time, it is infeasible to store all the massive data and then use straightforward methods such as sorting in an offline manner, but only feasible to process data in a *single pass*. Traditional sampling methods, such as uniform packet (or flow) sampling, record a small set of packets (or flows) to reduce memory usage and processing time. However they usually have to employ a low sampling rate and thus bring significant errors [11]. On the other hand, streaming algorithms [27] record a small amount of information called *digest*[2] for each piece of data, and can derive much more accurate answer from the digest.

2.2. Estimating flow statistics

For the sake of accurately catching top users, we must be able to count the number of specific flows for each user.

The simplest form of this problem is to count the total number of flows. On high speed links, sampling has to be used if we simply keep per flow information in a hash table (i.e., in Netflow [7] or Snort [32]), since hash table is both memory and time consuming [28,8]. Simple packet sampling will miss many small flows [12], while simple flow sampling must employ a very low sampling rate, since the worst case expense of accessing a hash table may be very high, and a large flow, if sampled, can cause continuous accesses.[3] On the other hand, if we regard flows as distinct elements and packets as their duplications, it is just a cardinality estimation problem that has been extensively studied in the literature of streaming algorithms. Besides some theoretical work [1,18], two families of algorithms, including the "bitmap sketch"[35,15,31] and the "LogLog sketch" [13,16], which we will denote by flow counting (FC) sketches, are the most efficient (for processing) and accurate (for estimation) in practice.

Although very important, counting small flows is far less studied due to its difficulty [29]. Theoretically speaking,[4] the total number of flows falls into the cash register model since a new packet will never decrease that number, while the number of small flows falls into the strict turnstile model since a new packet may decrease the number of small flows when a flow size changes from q to $q+1$. Some algorithms [10,29,21] estimate the flow size distribution but most of them will not perform well here since they do not focus on small flows. One algorithm [21], although is experimentally accurate, provides no theoretical analysis.

There are also works estimating other useful flow statistics on single or multiple streams, including per flow size [24], flow entropy [22,36], traffic and flow matrices [38], to name but a few. Although not directly related to our problem, they are worth further exploration.

[2] In the literature, this information is called *synopsis*. We use *digest* to be coherent with the name of our data structure.

[3] Using CAM, although can guarantee bounded access time, is much more expensive.

[4] Refer to [27] for the theoretical models.

2.3. Finding frequent or top elements

Another closely related and extensively studied problem is to ask for elements of top or high frequency, so called "heavy hitters". Given ϵ as a user-defined error, it asks a list of (at most) k elements such that each element E_i in the list has frequency $F_i > (1-\epsilon)F_k$, where F_k is the frequency of the kth most frequent element or just a threshold. A lot of algorithms [5,25,26,9] have been proposed, but all of them can only work on one level of aggregation, which is to say, they can find top hosts (or elephant flows) having a large number of packets, but are not directly applicable to our problem of finding hosts with a large number of (small) flows, since flow itself is one level of aggregation.

2.4. Catching top users

We have discussed the shortcomings of simply storing per flow information or sampling, and now we introduce some related non-trivial algorithms.

The distinct-count (DC) sketch [17] designs special hash tables and collects flows without hash collision as samples. It is indeed a kind of flow sampling, and can be used for catching both top spreaders and scanners. Since flows are allowed to co-exist in the same bucket, collision resolution is not needed and the speed of the DC sketch can be guaranteed. However, its accuracy is inherently limited by sampling.

Zhao et al. [37] proposed streaming algorithms for catching "super spreaders" who spread a large number of flows (no explicit threshold was considered). Their first algorithm uses a filter similar to bloom filter [3], and their second algorithm uses an array of bitmap sketches, which we will explain in more detail in Section 3.1. They evaluated on cardinalities of 15–200, and concluded that the filter based algorithm was less accurate and might be used on low speed links, while the bitmap sketches based one was more accurate and could work with speed up to 40 Gbps using SRAM. For catching the top-k spreaders, it is straightforward to use the FC sketches (i.e., bitmap/LogLog sketches) with a heap, as done in some recent works [33,19]. However, they do not have good scalability with respect to traffic skewness and link speed, as we will analyze in Section 3.4.

Other special sampling methods [34,20,4] have also been proposed to find hosts with cardinalities above an explicit threshold or within a specific range, but they often tolerate a high cardinality error (i.e., $\geqslant 20\%$) since they focus on finding hosts but not ranking them. Besides, they cannot be easily extended for catching scanners.

We propose streaming algorithms to catch top scanners, and address several challenges of scalability, including: *traffic scale, skewness, speed*, and *result accuracy*. The algorithms can also be used for catching spreaders without any trouble. Although the DC sketch has the same capabilities, we will show that our framework performs several orders of magnitude better, since they use collision-free samples, while we dig out information hidden in collisions.

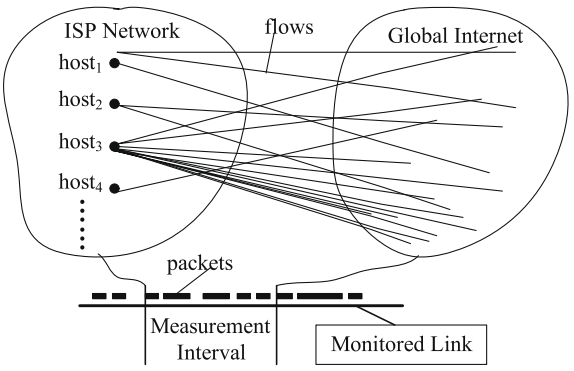

Fig. 1. User communication senario.

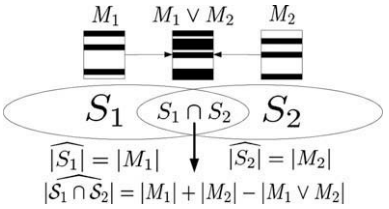

Fig. 2. Set and sketch operations.

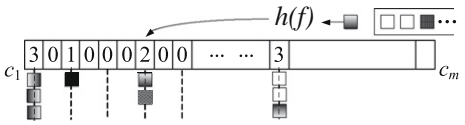

Fig. 3. Sketch for Algorithm A.

3. Algorithm A – sketches based

In this section, we extend the flow counting sketches based idea into our first algorithm for catching scanners.

3.1. Composing FC sketches for spreaders

First let us review the idea of composing FC sketches for catching spreaders, using the bitmap sketch as an example. When n flows are uniformly mapped into an m bit bitmap so that packets of the same flow set the same bit to "1", the untouched bits in the bitmap is expected to be $y_0 = m(1-1/m)^n \approx me^{-n/m}$, so n can be estimated by $m\ln(m/y_0)$. A naive solution for catching top users is to maintain a FC sketch for each user [14,4], but it does not scale well with the number of users. Instead, a fixed number of sketches of the same size can be shared by all users as follows [38,33,19]:

They compose d equal-sized sketches $(M_1, M_2, \ldots, M_d)$ and ℓ independent uniform hash functions $(h_1, h_2, \ldots, h_\ell)$. A packet with user ID s is mapped to the same position of ℓ sketches, namely, $M_{h_1(s)}, M_{h_2(s)}, \ldots, M_{h_\ell(s)}$, and updates each of them. Assume the flow count estimated from sketch M_i is $|M_i|$, and the set of users that are mapped to M_i is $\mathcal{S}_i$, having $|\mathcal{S}_i|$ flows in total, then $|\mathcal{S}_i|$ can be estimated by $\widehat{|\mathcal{S}_i|} = |M_i|$. Given two sketches M_1 and M_2, the total flow count of $|\mathcal{S}_1 \cup \mathcal{S}_2|$ can be estimated by $\widehat{|\mathcal{S}_1 \cup \mathcal{S}_2|} = |M_1 \vee M_2|$, where $M_1 \vee M_2$ is the result of combining the two sketches M_1 and M_2 by *bit-wise OR* (or by *entry-wise maximum* if using the LogLog sketch). Then, as illustrated in Fig. 2, we have the estimator for set intersection:

$$\begin{aligned}\widehat{|\mathcal{S}_1 \cap \mathcal{S}_2|} &= \widehat{|\mathcal{S}_1|} + \widehat{|\mathcal{S}_2|} - \widehat{|\mathcal{S}_1 \cup \mathcal{S}_2|} \\ &= |M_1| + |M_2| - |M_1 \vee M_2|,\end{aligned}$$

and this can be easily generalized to ℓ sketches. Denoting by $||\mathcal{S}||$ the total number of users, if $d^\ell \gg ||\mathcal{S}||$, then the probability that any two users are mapped to the same ℓ sketches is very small.[5] So with high probability, we have $\mathcal{S}_{h_1(s)} \cap \mathcal{S}_{h_2(s)} \cap \ldots \cap \mathcal{S}_{h_\ell(s)} = \{s\}$, and

$$\widehat{|\{s\}|} = |\mathcal{S}_{h_1(s)} \cap \widehat{\mathcal{S}_{h_2(s)}} \cap \ldots \cap \mathcal{S}_{h_\ell(s)}|,$$

which means the flow count of user s can be derived from the ℓ sketches $M_{h_1(s)}, M_{h_2(s)}, \ldots, M_{h_\ell(s)}$ which s is mapped to. This nice property indicates that d sketches can be shared among many user, instead of one sketch per user.

We can also get the estimator for set difference:

$$\widehat{|\mathcal{S}_1 \setminus \mathcal{S}_2|} = \widehat{|\mathcal{S}_1 \cup \mathcal{S}_2|} - \widehat{|\mathcal{S}_2|} = |M_1 \vee M_2| - |M_2|,$$

and this can be used to count attack flows with no corresponding flows of reverse direction, i.e., SYN attack flows.

3.2. Sketch for counting small flows

We tailor the sketch for estimating flow size distribution [21] to count small flows, and derive new analytical results. Consider an array of m counters, $c_1, c_2, \ldots, c_m$, all being initialized to zero. We use a uniform hash function $h : f \to [1 \ldots m]$ to map each flow f (and all its packets) to a position $h(f)$, and use c_j to record the number of packets mapped to position j. As shown in Fig. 3, each square represents a packet, and the different gray patterns represent different flows.

After recording all packets, we use y_k to represent the total number of counters with value k. So there are y_0 counters with value zero, y_1 counters with value one, and so on. In addition, we use n to represent the total flow count, and use n_k to represent the number of flows with exactly k packets. These notations are summarized in Table 1.

Counting singleton flows. Since a counter c_j has value one if and only if exactly one singleton flow but no other flow maps into position j, this probability is expected to be $E(\frac{y_1}{m}) = \binom{n_1}{1}\frac{1}{m}(1-\frac{1}{m})^{n_1-1}(1-\frac{1}{m})^{\sum_{i\geqslant 2} n_i} \approx \frac{n_1}{m}e^{-n/m}$, and $\hat{n}_1 = y_1 e^{n/m}$ can be used as an estimator of n_1, which was evaluated experimentally in [21]. We give the following new theoretical results:

Lemma 1. *When n and m simultaneously approach infinity,*[6] *we have*

[5] This probability is $1 - \binom{d^\ell}{||\mathcal{S}||}||\mathcal{S}||!/(d^\ell)^{||\mathcal{S}||}$.

[6] Strictly speaking, when $\lim_{m\to\infty} n^2/m > 0$.

Table 1
Notations in Algorithms A&B.

m	the number of counters
h	the hash function mapping flow ID's to counters
y_k	the number of counters with value k
$y_k(t)$	similar to the definition of y_k, except right before The tth packet arrives
$\mathscr{F}$	the set of all flows
n	the total number of flows, $n = \lvert\mathscr{F}\rvert$
n_k	the number of flows with exactly k packets
α	loading factor, $\alpha = n/m$
α_k	$\alpha_k = n_k/m$
c_j	the value of the jth counter, and is also the number of packets mapped to position j
$c_j(t)$	similar to the definition of c_j, except right before the tth packet arrives
f_x	the xth packet of flow f
e_t	the tth packet among all packets
v_t	the counter value that e_t sees

$$\lim_{m\to\infty} E(y_0) = me^{-\alpha}, \lim_{m\to\infty} E(y_1) = n_1 e^{-\alpha},$$

where $\alpha = \lim_{m\to\infty} n/m$. *Furthermore,* y_0 *and* y_1 *are the MLEs of* $E(y_0)$ *and* $E(y_1)$, *respectively.*

Then we can get

Conclusion 1. The number of singleton flows, n_1, can be estimated by an asymptotically unbiased MLE:

$$\hat{n}_1 = m \times y_1/y_0. \tag{1}$$

The variance is approximately

$$n_1((e^{\alpha} - 2\alpha)\alpha_1 + e^{\alpha} - 1),$$

where $\alpha_1 = \lim_{m\to\infty} n_1/m$.

The proofs for Lemma 1, Conclusion 1 and more properties of y_1 can be found in our technical report [30]. Although we do not have a closed form to bound the error caused by the above approximation of the variance, the error is neglectable based on our experiments.

So if we remember the number of counters with value zero and one, namely, y_0 and y_1, we can estimate the number of singleton flows, namely, n_1. Each counter only needs two bits, since any value greater than one can be remembered as two. Next, we generalize this result to flows with q packets.

Counting small flows. Using this sketch, an EM (Expectation–Maximization) algorithm [21] has been proposed to estimate n_k, which is slow and can only guarantee a local optimal result. Here we derive a simpler procedure for estimate n_q for a small q. Use $\mathscr{B}_q$ to denote the set of different flow set patterns that have exactly q packets, and use β_i $(1 \leqslant i \leqslant |\mathscr{B}_q|)$ to denote a specific flow set pattern in $\mathscr{B}_q$, so there are totally q packets in β_i, and a counter with a value of q will correspond to some β_i. Suppose β_i is composed of $\beta_{i(1)}$ flows of size one, $\beta_{i(2)}$ flows of size two, …, $\beta_{i(q)}$ flows of size q, and no flow of size more than q, we can write $\beta_i = \langle \beta_{i(1)}, \beta_{i(2)}, \ldots, \beta_{i(q)}, 0, \ldots\rangle$ and order them: $\beta_1 = \langle q, 0, \ldots, 0, 0, \ldots\rangle$ corresponding to q flows of size one, $\beta_2 = \langle q-2, 1, 0, \ldots, 0, 0, \ldots\rangle$ corresponding to $q-2$ flows of size one and one flow of size two, and $\beta_{|\mathscr{B}_q|} = \langle 0, 0, \ldots, 1, 0, \ldots\rangle$ (the qth element is 1) corresponding to one flow of size q. Denoting the number of counters with value x by y_x, and the number of flows of size x by n_x, when m is large, we have $E(\frac{y_q}{m}) = \sum_{i=1}^{|\mathscr{B}_q|} Pr(\beta_i)$, where $Pr(\beta_i)$, the probability that the flow pattern β_i appears, is

$$\prod_{d=1}^{q} C_{n_d}^{\beta_{i(d)}} \left(\frac{1}{m}\right)^{\beta_{i(d)}} \left(1-\frac{1}{m}\right)^{n_d-\beta_{i(d)}} \prod_{d>q} \left(1-\frac{1}{m}\right)^{n_d}.$$

Then by simple manipulation, we get

$$\frac{E(y_q)}{m} \approx \frac{E(y_0)}{m}\left[\sum_{i=1}^{|\mathscr{B}_q|-1}\left(\prod_{d=1}^{q-1}\frac{(n_d/m)^{\beta_{i(d)}}}{\beta_{i(d)}!}\right) + \frac{n_q}{m}\right],$$

so

$$n_q \approx m \times \left(\frac{E(y_q)}{E(y_0)} - \sum_{i=1}^{|\mathscr{B}_q|-1}\left(\prod_{d=1}^{q-1}\frac{(n_d/m)^{\beta_{i(d)}}}{\beta_{i(d)}!}\right)\right).$$

Since $\mathscr{B}_q$ can be pre-computed for any q, we get,

Conclusion 2. We can use an iterative equation

$$\hat{n}_q = m \times \left(\frac{y_q}{y_0} - \sum_{i=1}^{|\mathscr{B}_q|-1}\left(\prod_{d=1}^{q-1}\frac{(\hat{n}_d/m)^{\beta_{i(d)}}}{\beta_{i(d)}!}\right)\right) \tag{2}$$

to estimate $n_1, n_2, \ldots, n_q$ one by one.

For example, when $q = 1$, Eq. (2) degenerates to $m \times y_1/y_0$ in Conclusion 1; $\hat{n}_2 = m \times (y_2/y_0 - (\hat{n}_1/m)^2/2)$, and $\hat{n}_3 = m \times (y_3/y_0 - (\hat{n}_1/m)^3/6 - \hat{n}_1/m \times \hat{n}_2/m)$. Besides, each counter needs $\lceil \log(q+2)\rceil$ bits since any value greater than q can be recorded as $q+1$. We might expect the asymptotic maximum likelihood and unbiasedness of estimator (2), however, the accuracy will decrease when q becomes larger, since its error is bounded by errors of $\hat{n}_1, \ldots, \hat{n}_{q-1}$ and y_q/y_0. We will present its accuracy in the evaluation section.

Counting total flows. When using a single bit for each counter, the sketch falls back into a bitmap sketch, so the old algorithms [14,31] apply.

3.3. Composing sketches for scanners

We will use sketches composed of this kind of counter arrays in our framework. However, when d sketches are shared among many users, the relation between sketch operations and set operations, which is essential for deriving the small flow count for a specific user s, is not as trivial as that for spreaders (see Section 3.1). The final formulas

are simple but are not straightforward to derive, and we put them in our technical report [30] for brevity. Here we only note that sketches are combined by *counter-wise addition*.

3.4. Shortcomings of sketches based algorithms

Three shortcomings exist in sketches based algorithms. First, *it is impossible to recover user ID's from the composed sketches*. One remedy [37] is to explicitly store all user ID's but is memory and time consuming. Another method [33,19] is to use a heap to track only those top users but faces the second problem: *estimating flow count from sketches is not fast enough to keep up with the link speed*, since sketch combination is time consuming. At last, *the estimation error σ increases when traffic skewness decreases*, as suggested by the following lemma:

Lemma 2.

$$\sigma\left(\frac{|\mathcal{S}_1 \cap \widehat{\ldots} \cap \mathcal{S}_\ell|}{|\mathcal{S}_1 \cap \ldots \cap \mathcal{S}_\ell|}\right) \Big/ \sigma\left(\frac{|\widehat{\mathcal{S}_i}|}{|\mathcal{S}_i|}\right) \sim O\left(\frac{|\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_\ell|}{|\mathcal{S}_1 \cap \ldots \cap \mathcal{S}_\ell|}\right).$$

An intuitive but less rigorous explanation is as follows.[7] Given a bitmap sketch M_i of size m, the *relative standard deviation* $\sigma\left(|\widehat{\mathcal{S}_i}|/|\mathcal{S}_i|\right) \sim O(1/\sqrt{m})$ [35,13], and for the combination of two sketches, $\sigma\left(|\widehat{\mathcal{S}_1 \cup \mathcal{S}_2}|/|\mathcal{S}_1 \cup \mathcal{S}_2|\right) \sim O(1/\sqrt{m})$. Then

$$\begin{aligned}\sigma\left(\frac{|\mathcal{S}_1 \widehat{\cap} \mathcal{S}_2|}{|\mathcal{S}_1 \cap \mathcal{S}_2|}\right) &= \sigma\left(\frac{|\widehat{\mathcal{S}_1}| + |\widehat{\mathcal{S}_2}| - |\mathcal{S}_1 \widehat{\cup} \mathcal{S}_2|}{|\mathcal{S}_1 \cap \mathcal{S}_2|}\right) \\ &\leqslant \frac{|\mathcal{S}_1 \cup \mathcal{S}_2|}{|\mathcal{S}_1 \cap \mathcal{S}_2|} \times \left(\sigma\left(\frac{|\widehat{\mathcal{S}_1}|}{|\mathcal{S}_1|}\right) + \sigma\left(\frac{|\widehat{\mathcal{S}_2}|}{|\mathcal{S}_2|}\right) + \sigma\left(\frac{|\mathcal{S}_1 \widehat{\cup} \mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|}\right)\right) \\ &\sim O\left(\frac{|\mathcal{S}_1 \cup \mathcal{S}_2|}{|\mathcal{S}_1 \cap \mathcal{S}_2|}\right) \times \sigma\left(\frac{|\widehat{\mathcal{S}_i}|}{|\mathcal{S}_i|}\right).\end{aligned}$$

Generalizing it to ℓ sketches proves Lemma 2.

So even the flow count of s can be derived form the ℓ sketches it maps to as $\widehat{|s|} = |\mathcal{S}_1 \cap \mathcal{S}_2 \widehat{\cap} \ldots \cap \mathcal{S}_\ell|$, Lemma 2 tells us that the estimation error of $\widehat{|s|}$ will increase with $|\mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_\ell|$. $|\mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_\ell|$ is just the total flow count of all users mapped to any of the ℓ sketches, and these flows will *interfere* with flows of s. Intuitively, the expected number of *interfering* flows is approximately $\ell^2 \times n/d$, which will be greater than the flow count of a top user if d is small and the traffic is not very skewed. For example, with $n = 10^6, d = 1000$ and $\ell = 3$, $\ell^2 \times n/d$ is 9000, and the estimation error will be high unless $|s|$ is much larger than 9000, and a non-top user spreading only a few flows may be wrongly estimated to be spreading a large number of flows. Even this error magnification effect may possibly be reduced by extending some special sketch [6], the other two shortcomings still exist. These are inherent shortcomings of sketches based algorithms (including ours), and we will revisit them in our framework.

[7] A more rigorous proof for composing bitmap sketches can be found in [6].

4. Algorithm B – filter based

4.1. Counting small flows

Now we introduce our second algorithm for counting small flows. An array of m counters are used again and the update process of the counters, instead of only their final values in algorithm A, is investigated. We use the same notations as before, except that we use $y_k(t)$ and $c_j(t)$ to explicitly denote the number of counters with value k and the value of the jth counter *right before the tth packet arrival*, respectively. As shown in Fig. 4, B_1, the tth packet, sees a counter value of two. Gray squares represent packets that have arrived before B_1, squares with characters represent forthcoming packets, and different patterns or characters represent different flows. For clarity we have omitted the uniform hash function that maps flows to counters.

First we establish a simple lemma stating that, the counter value that any packet sees is independent of the arriving order of its preceding packets.

Lemma 3. *Assuming the flow ID of the tth packet is f, $c_{h(f)}(t)$, the counter value it sees, is independent of the arriving order of its preceding $t-1$ packets.*

Proof. It holds simply because $c_{h(f)}(t)$ only depends on the total number of packets mapped into position $h(f)$ before time t, but not their order. □

Let us denote the set of flows by $\mathcal{F}$, and denote the xth packet of flow f by f_x. We also denote the tth packet among all packets by e_t, which sees value v_t, so the event "$\exists f \in \mathcal{F}, e_t = f_x$" (abbreviated to "$e_t = f_x$") means "the tth packet is the xth packet of some flow f, and $v_t = c_{h(f)}(t)$". Let $I(e_t = f_x)$ be an indicator function which equals 1 if the tth packet really is the xth packet of some flow and 0 otherwise. The total number of flows is just $n = \sum_t I(e_t = f_1)$, that is the number of the packets, of which each is the first packet of some flow. The number of small flows of size no more than q is $\sum_{k=1}^{q} n_k = \sum_t I(e_t = f_1) - \sum_t I(e_t = f_{q+1})$, where n_k is the number of flows with k packets.

If there is no mapping collision between flows, the first packet of a flow will always see a counter value of zero, and the xth packet of a flow will always see a counter value of $x-1$. That is, $I(e_t = f_x) = I(v_t = x-1)$. So $\sum_{k=1}^{q} n_k =$

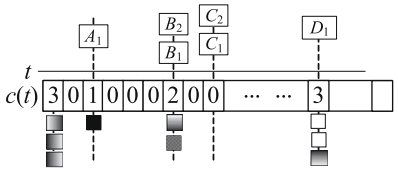

Fig. 4. Algorithm B.

$\sum_t I(e_t=f_1)-\sum_t I(e_t=f_{q+1})=\sum_t I(v_t=0)-\sum_t I(v_t=q)$, which means, we can get the total number of flows by counting packets seeing value zero, and get the number of small flows of size no more than q by deducting the number of packets seeing value q.

However there do exist collisions and we have to do a deeper analysis on $I(e_t=f_1)$ and $I(e_t=f_{q+1})$.

Theorem 1. *The conditional probability that the tth packet e_t sees a counter value of z, given that e_t is the xth packet of some flow, is:*

$$Pr(v_t=z|e_t=f_x)=\begin{cases}0, & \text{if } z<x-1;\\ y/m, & \text{if } z\geqslant x-1 \text{ and } x=1;\\ p, & \text{if } z\geqslant x-1 \text{ and } x>1;\end{cases}$$

where $y=y_{z-x+1}(t)$ is the number of counters with value $z-x+1$ right before this packet arrives, $(y-1)/m\leqslant p\leqslant (y+1)/m$ and with high probability, $p=y/m$. Particularly, for $x=1$, that is for the first packet of each flow, we have $Pr(v_t=z|e_t=f_1)=y_z(t)/m$.

Proof

(1) Since the preceding $x-1$ packets belonging to the same flow as e_t map into the same position, z is at least $x-1$, so the probability is always zero when $z<x-1$.
(2) For $x=1$, which means e_t is the first packet of some flow, since the position mapping is uniform and is independent of any proceeding packets, the probability that it sees value z is just the percentage of counters with value z at that time, that is, $y_z(t)/m$.
(3) For $x>1$, however, the mapping of B_x is dependent on B_1 and the probability is not simply $y_z(t)/m$. Let us focus on the counter which $B_1, B_2, \ldots$ are mapped to, as shown in Fig. 5. Due to collisions, there may be some other flows (gray squares) mapped to that counter.

In the first row, the tth packet B_x is the xth packet of flow B, and it sees a counter value of z'. At the first glance, it is hard to tell what the probability of $z'=z$ is. However as we know, the probability only depends on the counter values at that time, and by Lemma 3, we can always reorder its preceding packets without changing the counter values. So we can look for a particular ordering to help the calculation. Specifically, we put all the preceding $x-1$ packets of the same flow right before this packet,

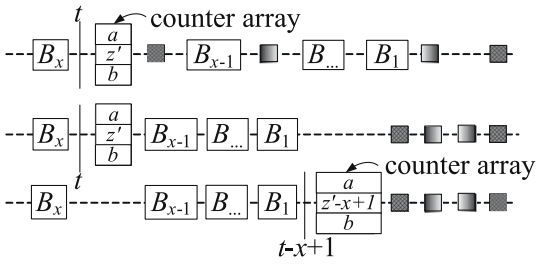

Fig. 5. Reordering packets.

as shown in the second row of Fig. 5, then B_1, the first packet of flow B, would have seen a counter value of $z'-x+1$, as shown in the third row. *For this particular reordering*, we have $Pr(v_t=z|e_t=f_x)=Pr(c_{h(f)}(t-x+1)=z-x+1|e_t=f_x)$, which means, the probability that B_x sees value z equals the probability that B_1 sees value $z-x+1$.

It should be noted that *all the other counters remain unchanged after this reordering*. So if in the first and second row there are $y=y_{z-x+1}(t)$ counters having value $z-x+1$ right before B_x (the tth packet) arrives, then in the third row there must be at lease $y-1$ and at most $y+1$ counters having value $z-x+1$ right before B_1 arrives, since no other counter except the one B_1 (or B_x) is mapped to may change its value.[8] That is to say, $(y-1)/m\leqslant Pr(c_j(t-x+1)=z-x+1|e_t=f_x)\leqslant (y+1)/m$, and hence $(y-1)/m\leqslant p\leqslant (y+1)/m$. And with high probability, $p=y/m$. □

In one word, Theorem 1 tells us that, the conditional probability that the xth packet of a flow sees value z is approximately the percentage of counters having a value of $z-x+1$ when that packet arrives.

Then we can rewrite the probability that the tth packet is the qth packet of some flow as follows:

Theorem 2. *$Pr(e_t=f_q)$ can be decomposed as*

$$Pr(e_t=f_q)\approx\sum_{i=0}^{q-1}Pr(v_t=i)\times Y_{i,q}(t), \tag{3}$$

where $Y_{i,q}(t)$ is a fixed polynomial expression of $y_1(t)/y_0(t)$, $y_2(t)/y_0(t),\ldots,y_{q-i-1}(t)/y_0(t)$, and $m/y_0(t)$.

Proof. We prove by induction.

(1) For $q=1$, since $Pr(v_t=0,e_t=f_1)=Pr(v_t=0)$,

$$Pr(e_t=f_1)=\frac{Pr(v_t=0,e_t=f_1)}{Pr(v_t=0|e_t=f_1)}=Pr(v_t=0)\times\frac{m}{y_0(t)},$$

so the theorem holds for $q=1$.

(2) For $q>1$, we have

$$\begin{aligned}Pr(e_t=f_q)&=\frac{Pr(v_t=q-1,e_t=f_q)}{Pr(v_t=q-1|e_t=f_q)}\\ &\approx\frac{Pr(v_t=q-1)-\sum_{x=1}^{q-1}Pr(v_t=q-1,e_t=f_x)}{y_0(t)/m}\\ &=\frac{Pr(v_t=q-1)-\sum_{x=1}^{q-1}Pr(v_t=q-1|e_t=f_x)\times Pr(e_t=f_x)}{y_0(t)/m}\\ &\approx Pr(v_t=q-1)\times\frac{m}{y_0(t)}-\sum_{x=1}^{q-1}Pr(e_t=f_x)\times\frac{y_{q-x}(t)}{y_0(t)},\end{aligned}$$

where we approximate[9] $Pr(v_t=z|e_t=f_x)$ by $y_{z-x+1}(t)/m$. By induction, our theorem holds for any q. □

[8] If it happens to change *from* $z-x+1$ to any other value, we have $y-1$. If it happens to change from z *to* $z-x+1$, we have $y+1$. Else we have y.

[9] Here we neglect the small error of at most $1/m$ to simplify the analysis. A more rigours analysis without approximation can be found in our technical report [30]. Both our analysis and evaluation show that the degrade of the accuracy can be safely neglected.

By applying an expectation to Eq. (3), we have:

Corollary 1

$$E(I(e_t = f_q)) \approx \sum_{i=0}^{q-1} E(I(v_t = i)) \times Y_{i,q}(t).$$

The exact expressions of $Y_{i,q}(t)$'s are fixed and can be pre-computed. Particularly, for $q = 1$ and 2,

$$E(I(e_t = f_1)) = E(I(v_t = 0)) \times m/y_0(t), E(I(e_t = f_2)) \approx \frac{E(I(v_t = 1))}{y_0(t)/m} - \frac{E(I(v_t = 0))}{y_0(t)/m} \times \frac{y_1(t)}{y_0(t)}.$$

Theorem 2 and its corollary tell us that, *the counter values, which are the bare knowledge we have, can indicate the position of a packet inside a flow*. Now let us use this information to count specific flows.

Counting singleton flows. The number of flows with only one packet is

$$\begin{aligned} n_1 = E(n_1) &= E\left(\sum_t I(e_t = f_1) - \sum_t I(e_t = f_2)\right) \\ &= \sum_t E(I(e_t = f_1)) - \sum_t E(I(e_t = f_2)) \\ &\approx \sum_t \left(\left(\frac{y_1(t)}{y_0(t)} + 1\right) \times \frac{E(I(v_t = 0))}{y_0(t)/m}\right) - \sum_t \frac{E(I(v_t = 1))}{y_0(t)/m} \\ &= E\left[\sum_{t:v_t=0} \left(\left(\frac{y_1(t)}{y_0(t)} + 1\right) \times \frac{m}{y_0(t)}\right) - \sum_{t:v_t=1} \frac{m}{y_0(t)}\right], \end{aligned}$$

where the approximation is due to substituting $E(I(e_t = f_1))$ and $(I(e_t = f_2))$. Now we have decomposed n_1 into the expression of $y_0(t), y_1(t)$ and v_t, which are all observable: the number of counters with value zero and one, and the value that packet e_t sees. Due to the unbiasedness of expectation, we have

Conclusion 3. If the tth packet sees a counter value of zero, add $(\frac{y_1(t)}{y_0(t)} + 1) \times \frac{m}{y_0(t)}$ to the singleton flow count. If it sees a counter value of one, deduct $\frac{m}{y_0(t)}$ from the singleton flow count. This procedure gives us an asymptotically unbiased estimator for singleton flow count:

$$\hat{n}_1 = \sum_{t:v_t=0} \left(\left(\frac{y_1(t)}{y_0(t)} + 1\right) \times \frac{m}{y_0(t)}\right) - \sum_{t:v_t=1} \frac{m}{y_0(t)}. \quad (4)$$

Following a standard calculation, the variance of $\hat{n}_1$ can be derived as

$$\sum_{t:e_t=f_1} \left[\left(\left(\frac{y_1(t)}{y_0(t)} + 1\right)^2 + \frac{y_1(t)}{y_0(t)}\right) \times \frac{m}{y_0(t)} - 1\right] + \sum_{t:e_t=f_2} \left(\frac{m}{y_0(t)} - 1\right).$$

Counting small flows. Now for flows with at most q packets, we have

$$\begin{aligned} \sum_{k=1}^{q} n_k = E\left(\sum_{k=1}^{q} n_k\right) &= E\left(\sum_t I(e_t = f_1) - \sum_t I(e_t = f_{q+1})\right) \\ &= \sum_t E(I(e_t = f_1)) - \sum_t E(I(e_t = f_{q+1})). \end{aligned}$$

Then by decomposing $E(I(e_t = f_1))$ and $E(I(e_t = f_{q+1}))$ using our Corollary 1, we get

$$\begin{aligned} \sum_{k=1}^{q} n_k &= \sum_t \sum_{i=0}^{q} (E(I(v_t = i)) \times U_{i,q}(t)) \\ &= E\left(\sum_{i=0}^{q} \sum_{t:v_t=i} U_{i,q}(t)\right), \end{aligned}$$

where $U_{i,q}(t)$ is also a fixed polynomial expression of $y_1(t)/y_0(t), \ldots, y_{q-i}(t)/y_0(t)$, and $m/y_0(t)$. Again, we can draw a conclusion on counting small flows:

Conclusion 4. Asymptotically, $n_{1\sim q}$, the number of small flows that having no more than q packets, can be unbiasedly estimated by summing up $U_{i,q}(t)$'s for those packets that see a counter value of $i (i \leqslant q)$. That is,

$$\hat{n}_{1\sim q} = \sum_{i=0}^{q} \sum_{t:v_t=i} U_{i,q}(t). \quad (5)$$

It should be noted that the expressions of $U_{i,q}(t)$'s can be pre-computed, and the exact value of $U_{i,q}(t)$ can be computed online efficiently for a small q. The algorithms for pre-computing expressions of $Y_{i,q}(t)$'s and $U_{i,q}(t)$'s can be found in our technical report [30]. Although the exact variance of this estimator can be derived following a standard procedure, it is too long and tedious, and we only present our evaluation results in Section 6. However, we note here that, the variance is also a polynomial of $y_i(t)/y_0(t)$'s and $m/y_0(t)$, so if at least half of the counters are empty, that is $y_0(t) \geqslant m/2 \geqslant y_i(t), \forall i \geqslant 1$, we can derive a loose (but already good enough) bound on the variance. And as in algorithm A, only $\lceil \log(q+2) \rceil$ bits are needed for each counter, since we only need to record up to $q+1$.

Counting total flows. The total number of flows n can be rewritten as $n = E(n) = E(\sum_t I(e_t = f_1)) = \sum_t E(I(e_t = f_1)) = E(\sum_{t:v_t=0} m/y_0(t))$, where the last equation is due to substituting $E(I(e_t = f_1))$ by $E(I(v_t = 0)) \times m/y_0(t)$ and the linearity of expectation. So the total flow count n can be estimated unbiasedly by summing up $m/y_0(t)$'s for those packets that see a counter value of zero, that is to say, $\hat{n} = \sum_{t:v_t=0} m/y_0(t)$ is an unbiased estimator of n. This can be treated as a special condition of Eq. (5) for $q = 0$. Following a standard procedure, we can also derive its variance as $\sum_{t:e_t=f_1} (m/y_0(t) - 1)$. If we use a single bit for each counter, this estimator falls back to that of [37] but we build it upon theorems of conditional probabilities.

We call this algorithm *filter based*, since the counter array acts as a filter in the sense that, any packet sees a value greater than q will be filtered and cannot cause any action. Since a packet whose position inside a flow is after $q+1$ can only see a value greater than q, we know *a flow f can cause at most* $\min(q+1, ||f||)$ *updates, where* $||f||$ *is the number of packets in f*. This property can nicely eliminate the traffic bursts caused by large flows, which cannot handled by simple flow sampling.

4.2. Tracking top users

We can use a counter F_s to count the number of (total or small) flows for user s as follows: we record each arriving packet in our filter, and if its user ID is s, we translate it into

an update on its corresponding F_s by Eq. (5), and eventually get the flow count of s. However, since the number of users may be very large, we cannot afford one counter for each. The top element algorithms introduced in Section 2.3 naturally work with our filter, since the information provided by our filter is *an update on the flow count of a user* and we want to find *users with the largest flow count*. Only a small number of counters are needed, while previous algorithms that cannot get this flow update information have to keep a record for each user.

For our purpose, we derive an algorithm based on the theories proved in [26]. As show in Fig. 6, our tracker maintains a min heap P and a hash table H, both having c entries. The potential top users and their flow counts are stored in P, of which each entry has three fields: host ID s, flow count F_s, and error bound err_s on the count. The heap is dynamically updated according to the flow counts so the host s_{min} with the minimal flow count can be efficiently found. The hash table H is a quick index of P and stores pairs like (s, i_s), where the key s is a host ID, and the value i_s is the corresponding position of s in P. At most c users and their flow counts are tracked, using Algorithm 1 (UpdateT). When a count update Δ_s is issued for user s, if s is currently being tracked, its count F_s will be updated. Otherwise, s will replace s_{min} which has the minimal count value, F_s will be updated, and an error bound will be set. A small tracker can guarantee to track top spreaders if we assume our filter reports an accurate flow count, as state in the following theorem:

Theorem 3. *using* $c = \min\left(||\mathscr{S}||, \lceil \frac{n}{T} \rceil\right)$ *counters, any host with a total flow count of more than T is guaranteed to be tracked, where* $||\mathscr{S}||$ *is the number of hosts and n is the total number of flows. If the flow count of each host follows a Zipfian distribution with parameter* $\alpha(\alpha \geqslant 1)$*, then c can be reduced to* $\min\left(||\mathscr{S}||, O\left(\left\lceil \left(\frac{n}{T}\right)^{\frac{1}{\alpha}} \right\rceil\right), \lceil \frac{n}{T} \rceil\right)$.

Proof. This result has been proved for unit arrival of elements regardless of their orders [26], which corresponds to $\Delta_s = 1$ in our problem. Due to its independence of update arriving orders, it can be generalized to any fractional increment values ($\Delta_s = y_0(t)/m$) by changing the unit scale, fractioning, and reordering. □

Algorithm 1. UpdateT(host s, flow count update Δ_s)

```
1   if (s exists in H) then
2       set F_s = F_s + Δ_s and update P to be in order;
3   update position i_s in H.
4   else
5       find the host s_min with the minimal count min;
6       replace s_min with s in P;
7       set F_s = min + Δ_s, err_s = min, and update P;
8       replace (s_min, i_smin) with (s, i_s) in H.
```

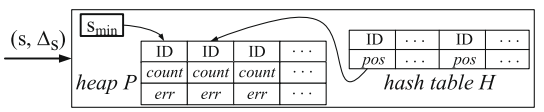

Fig. 6. Tracking top users.

However for counting small flows of scanners, we may have $\Delta_s = U_{i,q}(t) < 0$ when deducting some value from the flow count. Under this condition, we cannot do a straightforward substitution and get the result as using $c = \min\left(||\mathscr{S}||, \lceil \frac{n'}{T} \rceil\right)$ to catch top scanners having at least T small flows, where the total flow count (n) is substituted by the number of small flows (n'). Intuitively, consider such a worst case scenario for catching singleton scanners where there are two hosts X and Y. X has 10 singleton flows while Y has 20 large flows, so $n' = 10$. T is set to 10 to catch the scanner X, then $c = 1$ if we do a simple substitution. Host Y sends its 20 large flows in such a way that, it sends the first packet of each of its 20 flows altogether and defers its remaining packets till the end. The temporary singleton flow count of Y can be raised to 20 and this kicks X out of the tracker forever. However, if we still use $c = \min\left(||\mathscr{S}||, \lceil \frac{n}{T} \rceil\right)$ counters, even any other host can temporarily get a large small flow count, this temporary count will be no more than its total flow count, and top scanners would not be kicked out. Although theoretically n can be much large than n', they are usually of the same order in real traffic, since most flows are small flows.

4.3. A simple comparison with Algorithm A

The estimator (5) of our filter is simpler than the estimator (2) of sketches, and its estimation error is expected to be smaller (we will see this in the evaluation section). The shortcoming of algorithm B lies in that, it needs $\log c$ time (in the worst case) to update its heap in tracker $\mathbb{T}$ thus cannot catch up with the wire-speed of 40 Gbps even using SRAM. So *flow sampling has to be used with the filter even most packets have already been filtered out*. 40 Gbps requires a sampling rate of around 1/16, which although is much higher than that of traditional sampling, will enlarge and even dominate the filter estimation error. For a small $q(q \leqslant 3)$, algorithm A can effectively help reduce the error.

5. Online framework

5.1. Framework overview

As shown in Fig. 7, our framework is composed of the three data structures we have proposed, and deals with all their shortcomings. **Filter** $\mathbb{F}$ uses the filter based algorithm B and translates packets into updates on flow counts. **Tracker** $\mathbb{T}$ uses the information provided by $\mathbb{F}$ to

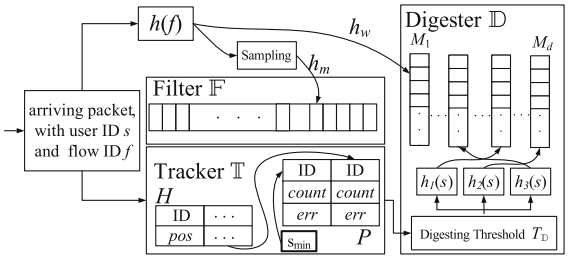

Fig. 7. Data structure for building blocks.

track the potential top users. **Digester** $\mathbb{D}$ uses the sketches based algorithm A and maintains flow digest for those potential top users being tracked by $\mathbb{T}$. Finally the information in $\mathbb{T}$ and $\mathbb{D}$ are combined for estimation. The idea behind our framework is to *dynamically track potential top users by roughly estimating their flow counts, and maintain digest of only those potential top users for accurate estimation*, so that (1) much fewer flows will be digested in $\mathbb{D}$ and the estimation by sketches will be nearly as accurate as one sketch per user, (2) query on sketches is not performed during update, (3) users' IDs are kept in $\mathbb{T}$, and (4) filter and sampling estimation error can be tolerated.

We note that the idea of *focusing on potential super/top users* also appears in [4] independently. However, their two-stage filtering can neither deal with traffic bursts of large flows nor be easily extended to support small flows.

Filter $\mathbb{F}$ is composed of m $\log\lceil q+2\rceil$-bit counters, tracker $\mathbb{T}$ has c entries, and digester $\mathbb{D}$ contains d sketches, each of which is composed of w $\log\lceil q+2\rceil$-bit counters. For catching scanners, q is just the threshold for small flows, and for catching spreaders, $q=0$. We use a hash function h_m to map a flow to a counter in $\mathbb{F}$, three hash functions h_i to map a host to three sketches, and another h_w to map a flow to a counter in each sketch.

5.2. Update and query algorithm

The update algorithm is very simple as shown in Algorithm 2 below. At the beginning of a measurement, counters in $\mathbb{F}$ and $\mathbb{D}$ are initialized to zero, entries in $\mathbb{T}$ are emptied, $y_1 \sim y_q$ are initialized to zero, and y_0 is set to m, the size of the filter. For each arriving packet, if it is from a potential top user (line 1),[10] three sketches in $\mathbb{D}$ are updated by simply incrementing a counter in each of them (line 2–4). $\mathbb{F}$ is updated by incrementing its counter at position $h_m(f)$ and adjusting two counters y_i and y_{i+1}, where i is the counter value at position $h_m(f)$ (line 6). Finally, tracker $\mathbb{T}$ is updated by the UpdateT algorithm, with a parameter $U_{i,q}$ computed online as described earlier (line 7).

Algorithm 2. Update(host ID s, flow ID f)

```
1  if (s in T) AND (F_s − err_s ⩾ T_D) then
2      for i = 1 to 3
3          if D_{h_i(s)}[h_w(f)] ⩽ q
4              D_{h_i(s)}[h_w(f)] + +;
5  if f should be sampled AND F[h_m(f)] ⩽ q then
6      i = F[h_m(f)]; F[h_m(f)] + +; y_i − −; y_{i+1} + +;
7      UpdateT (s, U_{i,q});
```

Since *each flow f can cause at most* $q+1$ *updates*,[11] long bursty traffic of large flows, alleviated by flow sampling, will not cause much overhead on $\mathbb{T}$. On the other hand, very few memory accesses in $\mathbb{F}$ and $\mathbb{D}$ are needed and can be parallelized by moderate hardware. The data structures can fit in a small piece of SRAM, and our framework is expected to work with a link speed as high as 40 Gbps, as shown in the evaluation section.

The query algorithm works as follows. For $q \leqslant 3$, use algorithm A and digester $\mathbb{D}$ to estimate the flow counts for the top-5k users[12] in $\mathbb{T}$ and add $T_{\mathbb{D}}/r$ to the results, sort, and output the adjusted top-k users with their flow counts. For $q > 3$, simply output the top-k users in $\mathbb{T}$ and their flow counts ($\mathbb{D}$ can be omitted).

The multi-resolution method [14,17,31] is orthogonal to our framework and they can be combined to handle a very large traffic scale. On the other hand, it is also easy to adapt our framework to count attack flows, since the counter arrays in both algorithm A and B naturally support set difference operation.

Table 2
statistics of the data for evaluation.

	#flows	#hosts	top	#s–f	#s–h	s–t
tr1	1.14 M	222 K	2810	647 K	152 K	2039
tr2	1.20 M	119 K	7316	730 K	78 K	7154
tr3	15.0 M	470 K	681 K	8.9 M	410 K	465 K
mix	1.06 M	100 K	83 K	672 K	67 K	60 K

6. Evaluation

In this section, we evaluate our framework using both real traces and synthetic data of different scale and skewness. Trace1 was captured on an ISP backbone link on April 2008 and a more skewed trace2 was captured on Nov 2008, and each of them lasted about two minutes. The synthetic data is generated using the Zipfian distribution, where the frequency of hosts with k flows is proportional to $1/k$ and the frequency of flows with k packets is proportional to $1/k^2$, and is mixed with simulated DDoS attack traces. We use them to compare our algorithms with some existing methods. We also generate a much longer trace3 from an one hour netflow trace captured on the uplink of a campus network on Jun 2007 to test the performance of our algorithms. Table 2 lists the numbers of flows and hosts, the number of flows of the top spreader, and the corresponding numbers for singleton flows and singleton scanners. For example, in trace1, there are 1.14 M flows of which 647 K are singleton flows, 222 K hosts of which 152 K spread singleton flows, and the top spreader spreads 2810 flows while the top singleton scanner spreads 2039 singleton flows. In these traces, *the top most scanner and around half of the top scanners are not among the top spreaders*, thus cannot easily be caught.

6.1. Setting parameters

Firstly we assume that n, the total flow count, can be predicated and use our trace1 as an example to discuss the parameters in our framework.

[10] The threshold $T_{\mathbb{D}}$ in line 1 of Algorithm 2 is to prevent users *who occasionally drop in the tracker* from being digested.

[11] Guaranteed by the test in line 3 and 5 in Algorithm 2.

[12] The value 5 is a little arbitrary but 3–7 all work well.

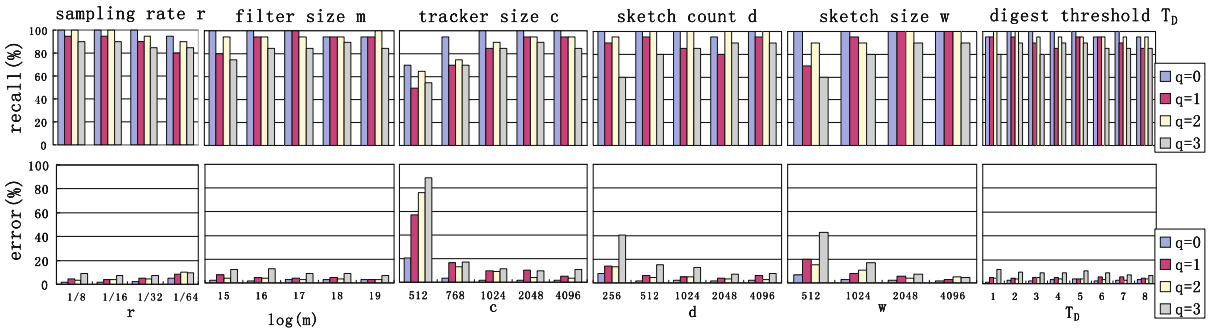

Fig. 8. Recall and flow count estimation error of top-20 users under different settings.

The first parameter we can set is the number of sketches d in digester $\mathbb{D}$, and $d = 1024$ is large enough since $d^{\ell} = 1024^3$ is much larger than any reasonable user count (recall Section 3.1). We set the tracker size $c = n/T$, where T is the threshold that we think such a flow count of a user is significant, as explained in Section 4.2. Indeed, c can be much smaller for more skewed data.

The sampling rate used by filter $\mathbb{F}$ is upper bounded by the update speed of our algorithms, whose bottleneck is the $\log c$ update time of tracker $\mathbb{T}$. A back-of-envelope calculation shows $r = 1/16$ is enough to handle tens of millions of flows under the 40 Gbps link speed when $T = 1000$, while a larger r up to 1 can be used on low speed links. With $r = 1/16$, the digesting threshold $T_{\mathbb{D}}$ can be set to a small number such as 2 or 3, which on one hand keeps packets of users who are occasionally tracked by $\mathbb{T}$ from being digested, while on the other hand keeps packets of real top users digested as much as possible.

For filter $\mathbb{F}$ to be accurate, its size m should be on the same order as rn, and for our traces with $n \approx 2^{20}$ and $r = 1/16$, m should be at least 2^{16} (counters). The digester sketch size w should be set to the same order as n/d, and here we use 1000 (counters). To keep almost half of the counters in $\mathbb{F}$ and $\mathbb{D}$ empty, m and w should be doubled to be 2^{17} and 2000, respectively.

We test our framework under different settings. The default parameters are: $r = 1/16, m = 2^{17}, d = 1024, w = 2048$ and $T_{\mathbb{D}} = 2$. For catching top spreaders we set $c = 1024$, and for top scanners we set $c = 2048$, corresponding to $T = 1000$ and 500, respectively. The total memory used is $m * \log\lceil q+2\rceil + c * 20 * 8 + d * w * \log\lceil q+2\rceil$ bits, which is 292 KB, 584 KB, 584 KB and 856 KB for $q = 0, 1, 2$ and 3, respectively. We vary one parameter at a time, and calculate the *recall* (true positive) of the top-20 users caught by our framework and the average *absolute relative error* of their flow counts estimation.

The results are depicted in Fig. 8. The default setting, as shown in the horizontal center of each figure, uses limited space but works quite well. The top-20 spreaders are almost always caught (at most one missed), and the estimation error is less than 3%. Sometimes, at most two or three top scanners are missed, but it is reasonable since the difference between the total (small) flow counts of spreaders (scanners) ranked around 20 is only around 10, which is fairly small compared with their flow counts of greater than 1000. The estimation errors of scanners are all below 10%, and fall below 3% (except 5% for $q = 3$) using memory budget around 1MB. An interesting phenomenon is that, results for $q = 2$, that is catching the top scanners spreading flows of one or two packets, is often as good as $q = 0$ and even better than $q = 1$, and we leave this to future studies.

We can see that, for $q \leqslant 3$, decreasing the sampling rate (r), the filter size (m), and the digester threshold ($T_{\mathbb{D}}$) have little impact on the result, since the accuracy is guaranteed by digester $\mathbb{D}$, which does not use sampling and can tolerate moderate error of filter $\mathbb{F}$. On the other hand, the tracker size (c), the digester sketch count (d) and size (w) do affect the result if they are too small, justifing our earlier analysis and arguments.

In practice, n is hard to predicate accurately and we usually have a pre-allocated memory budget. Following the above arguments, the memory used by each module should be $2r\log\lceil q+2\rceil \times n$, $\frac{160}{T} \times n$, and $2\log\lceil q+2\rceil \times n$, respectively, where r, q and T can be regarded as constants. So *we only need to allocate the memory to each module in propotion*. Furthermore, accompanied with a standard multi-resolution sampling method [15], traffic scale of orders of magnitude larger can also be handled appropriately.

6.2. Catching top spreaders

For catching top spreaders, we compare our framework with the "flow counting (FC) sketches + heap" based method and the filter based method (equipped with an additional small "tracker" to avoid storing all user ID's, and using a flow sampling rate $r = 1/16$). The second method is essentially the "filter + tracker" parts in our framework. For our method, we set $w = 1024, 2048, 4096, 8192, m = 2^{16}$ and set other parameters to their defaults. The memory used is 156 KB, 284 KB, 540 KB and 1052 KB, respectively, and we let the other two methods use the same amount of memory. Their recall and error on trace1 are shown in the left two sub figures in Fig. 9, where the results of "filter + tracker" without sampling are also presented for comparison.

We can see that, the error of FC sketches (the 4th bar) is high, especially when using limited memory (the recall can be lower than 50%, and the flow count estimation error can be above 50%), which justifies Lemma 2. Its relatively good

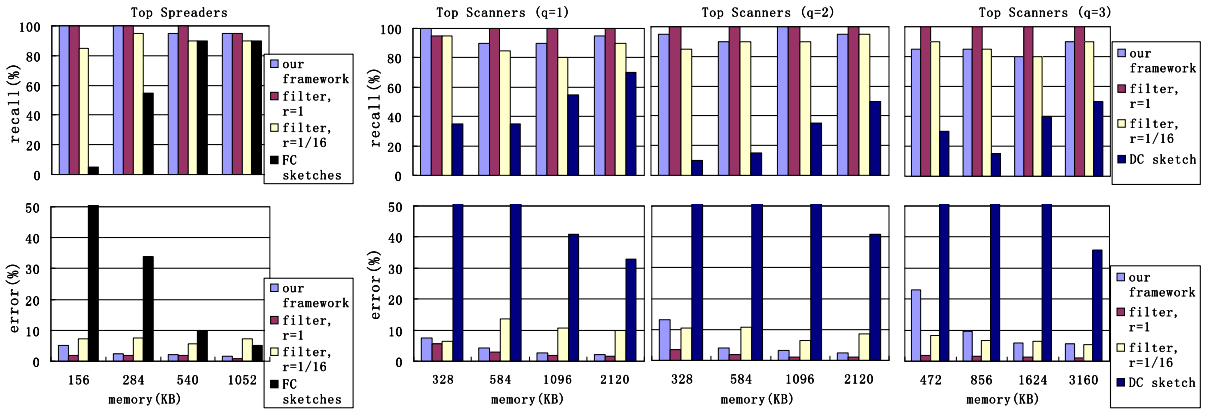

Fig. 9. Recall and flow count estimation error of top-20 users by different algorithms.

accuracy in Zhao et al. [37] is due to a much larger number of small sketches ($d = 16$ K and $w = 64$) for a small traffic scale (100 K flows), and only spreaders having 15–200 flows were evaluated. We can also see that the 10% error of filter with sampling (the 3rd bar) is dominated by sampling and hardly drops with more memory.[13] Using less than 300 KB memory, the estimation error of our framework (the 1st bar) drops quickly to below 3%, which is several times smaller than that of filter with sampling and an order of magnitude better than FC sketches. A deeper analysis shows that less than 1/10 flows leave information in our digester, so the error is effectively reduced, since the expected number of *interfering* flows drops to one tenth. For trace2 and synthetic data, the first two methods work better since the top users have much more flows, but are still much worse than our framework. We note that, filter without sampling (the 2nd bar) performs even better, indicating that, on relatively low speed links (e.g., 1 Gbps) we can simply use "filter + tracker" without sampling or with a very high sampling rate.[14]

6.3. Catching top scanner

For catching top scanners, we use the same parameters as above and the memory used varies from 328 KB to 2120 KB. Since no previous work directly solves this problem, we compare with an appliance of the Distinct-Count (DC) sketch [17]. Its first level is a multi-resolution sampling, and the second level uses r parallel hash tables with s buckets each. We use $r = 3$ as in the original work, and to guarantee an (ϵ, δ) estimation, s should be set to at least $\Theta(n \log(N/\delta)/(T\epsilon^2))$ to ensure that enough samples can be collected, where n and n are the total flow and packet counts, and T is the threshold to be considered as significant. Each bucket in the hash tables is composed of $x + 1$ specially designed counters where x is the number of bits in a flow ID, so that when flows without hash collision are collected as samples, their flow ID's can be recovered from the $x + 1$ counters. For each arrival, $r(x + 1)$ counters need to be updated. For a typical setting of $\epsilon = 1/3, \delta = 0.1, T = 1000, n = 10^6, N = 10 \times 2^{20}$ and $x = 100$, 120 million counters are needed in each table. So we allocate 10 times memory as used by our framework to each component of the DC sketch. We also compare with the method of "filter + tracker" with or without flow sampling. Their results on trace1 are depicted in the six sub figures in the right panel of Fig. 9, where the memory used by the DC sketch should be multiplied by 10.

We can see that the DC sketch (the 4th bar) does not perform well, since it is only appropriate for finding *significant anomaly* in highly skewed data, e.g., following a Zipfian distribution with parameter $\alpha > 1$ in their experiments, where n/T can be very small. For less skewed data, such as the small flow count of each host, it cannot find enough collision-free samples unless using a huge number of buckets. Instead, we dig out information hidden in collisions. Using around 1 MB memory, the estimation error of our framework (the 1st bar) is less than 3% for $q = 1$ and 2, and less than 6% for $q = 3$. Very few scanners are missed due to very small differences in flow counts, similar to that in catching spreaders. Filter with sampling (the 3rd bar) is beaten, since its error is still mainly due to sampling and barely decreases with more memory. For $q > 3$, since algorithm A does not help much, we only use "filter + tracker" and increase the memory linearly with $\log\lceil q + 2\rceil$ due to a larger counter size, and the performance only slowly degrades. Again, filter without sampling (the 2nd bar) performs the best (error $< 1\%$ using 1 MB memory), and can be used on low speed links. The results on other data are similar and they justify the usefulness of both algorithm A and B.

6.4. Varying traffic skewness

The accuracy of our framework on traffic of different skewness is shown in Fig. 10, where the same parameters as in Sections 6.2 and 6.3 are used. We can see, even without carefully tuning the parameters, our framework performs quite well on all traces when using only around 1 MB memory, and a little better on more skewed data,

[13] For comparison, using the same memory budget, the latest special sampling method [4] will produce an estimation error around 20%.

[14] For comparison, methods like Snort can only handle around 40 Mbps without sampling [28].

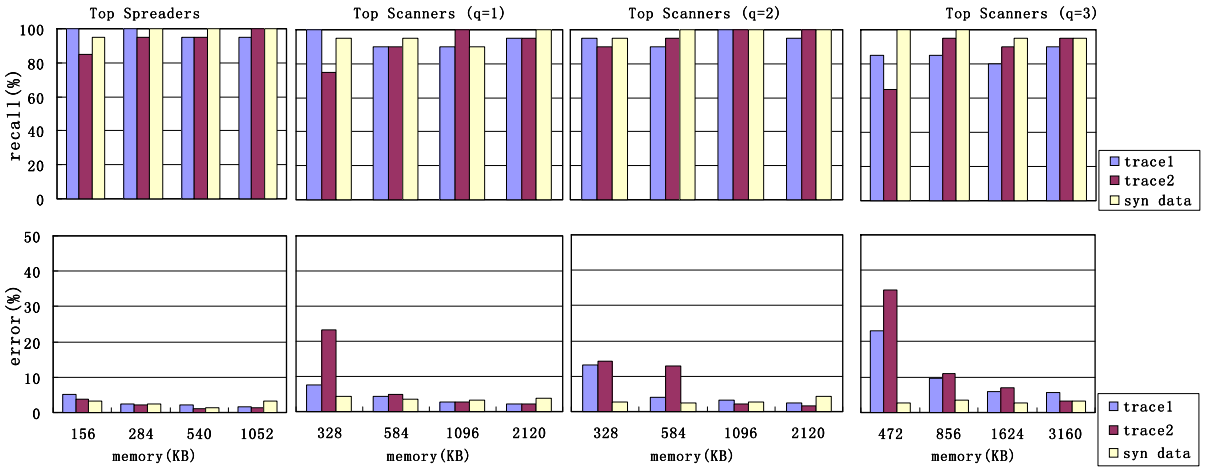

Fig. 10. Recall and flow count estimation error of top-20 users on different traffic.

since the top users spread more flows so that the estimation errors are reduced. We also note that, for highly skewed data, the sketches in digester $\mathbb{D}$ may be filled up and cannot be used for estimation, then the corresponding count in $\mathbb{T}$ will be used. On the other hand, even for highly skewed data, the other methods cannot perform as well as our framework.

6.5. Update and query speed

In Table 3, we list the update speed and query speed of the three different methods under different memory budgets, where the experiments were conducted on a single core of an Intel Core 2 1.86 G machine. The results are for $q = 1$ and trace1, while results for other q's and traces are similar. The update speed of the FC sketches based method is less than 100 Kpps and decreases with more memory since sketches need to be combined, and is close to previous results [19]. A query can be answered immediately since top users are stored in a heap. The DC sketch updates a constant number of counters for each arrival and achieves a constant update speed of around 200 Kpps, which is ten times faster than previous results [17] on an Intel PIII machine, and can support around 100 queries per second. An advanced DC sketch [17] with the same update speed can answer queries immediately, but has a penalty of doubled memory usage. Our framework, which only updates very few counters for each arrival, can achieve a constant update speed around 8 Mpps without any optimization nor special hardware, and is fast enough to support monitoring the bi-directional traffic on one 2.5 Gbps full-duplex link in the worst case. Around 1000 queries can be answered in one second when using 1 MB memory, and are enough for most practical applications.[15] Testing on the much longer trace3 with memory budget varying from 1 MB to 10 MB gets the same update speed. Since the latency of the DDR2 memory we use is around 100 ns, our framework is expected to handle 40 Gbps (125 Mpps under 40 Byte packets) by using SRAM of 5 ns latency.

Table 3
Performance of different algorithms.

	Update speed (M/s)			Query speed (K/s)		
Mem (KB)	584	1096	2120	584	1096	2120
Our fw	7.9	7.9	7.9	1.6	1.0	0.5
FC sketches	0.07	0.04	0.02	–	–	–
DC sketch	0.2	0.2	0.2	0.1	0.1	0.1

7. Conclusion

We propose a framework to catch a general types of top users. Our framework is based on combining sampling and streaming, as well as deterministic and randomized algorithms, which help each other to meet speed and resource requirements while maintaining good accuracy. Our framework addresses challenges of traffic scale, skewness and speed, uses very little resource, and achieves good accuracy even under the worst case. We believe it can be used in many practical online applications such as network monitoring and stream database.

Acknowledgement

Supported by Hong Kong RGC Grant NSFC-RGC N_CUHK414/06.

References

[1] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, J. Comput. Syst. Sci. 58 (1) (1999) 137–147.
[2] G. Armitage, Optimising online fps game server discovery through clustering servers by origin autonomous system, in: NOSSDAV '08, 2008, pp. 3–8.
[3] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[4] J. Cao, Y. Jin, A. Chen, T. Bu, Z. Zhang, Identifying high cardinality internet hosts, in: INFOCOM '09, 2009.

[15] The query speed actually decreases almost linearly with increased memory.

[5] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: ICALP '02, 2002, pp. 693–703.
[6] A. Chen, J. Cao, T. Bu, A simple and efficient estimation method for stream expression cardinalities, in: VLDB '07, 2007, pp. 171–182.
[7] Netflow, <www.cisco.com/web/go/netflow>.
[8] Netflow Performance Analysis, <www.cisco.com/en/US/tech/tk812/tech_white_papers_list.html>.
[9] G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, ACM Trans. Database Syst. 30 (1) (2005) 249–278.
[10] M. Datar, S. Muthukrishnan, Estimating rarity and similarity over data stream windows, in: ESA '02, 2002, pp. 323–334.
[11] N. Duffield, C. Lund, M. Thorup, Properties and prediction of flow statistics from sampled packet streams, in: IMW '02, 2002, pp. 159–171.
[12] N. Duffield, C. Lund, M. Thorup, Estimating flow distributions from sampled flow statistics, IEEE/ACM Trans. Netw. 13 (5) (2005) 933–946.
[13] M. Durand, P. Flajolet, Loglog counting of large cardinalities, in: ESA '03, 2003.
[14] C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice, ACM Trans. Comput. Syst. 21 (3) (2003) 270–313.
[15] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high-speed links, IEEE/ACM Trans. Netw. 14 (5) (2006) 925–937.
[16] P. Flajolet, Éric Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: AofA '07, 2007.
[17] S. Ganguly, M. Garofalakis, R. Rastogi, K. Sabnani, Streaming algorithms for robust, real-time detection of ddos attacks, in: ICDCS '07, 2007.
[18] P. Indyk, Stable distributions, pseudorandom generators, embeddings and data stream computation, J. ACM 53 (3) (2006) 307–323.
[19] K. Ishibashi, T. Mori, R. Kawahara, Y. Hirokawa, A. Kobayashi, K. Yamamoto, H. Sakamoto, Estimating top n hosts in cardinality using small memory resources, in: ICDEW '06, 2006, p. 29.
[20] N. Kamiyama, T. Mori, R. Kawahara, Simple and adaptive identification of superspreaders by flow sampling, in: INFOCOM '07, 2007, pp. 2481–2485.
[21] A. Kumar, M. Sung, J. Xu, J. Wang, Data streaming algorithms for efficient and accurate estimation of flow size distribution, SIGMETRICS Perform. Eval. Rev. 32 (1) (2004) 177–188.
[22] A. Lall, V. Sekar, M. Ogihara, J. Xu, H. Zhang, Data streaming algorithms for estimating entropy of network traffic, in: SIGMETRICS '06, ACM, New York, NY, USA, 2006, pp. 145–156.
[23] J.C. Louis, Outpost24 tcp vulnerability, <www.outpost24.com/news/news-2008-10-02.html>.
[24] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, A. Kabbani, Counter braids: a novel counter architecture for per-flow measurement, in: SIGMETRICS '08, 2008, pp. 121–132.
[25] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: VLDB '02, 2002, pp. 346–357.
[26] A. Metwally, D. Agrawal, A.E. Abbadi, An integrated efficient solution for computing frequent and top-*k* elements in data streams, ACM Trans. Database Syst. 31 (3) (2006) 1095–1133.
[27] S. Muthukrishnan, Data Streams: Algorithms and Applications, Now Publishers Inc., 2005.
[28] N. Paulauskas, J. Skudutis, Investigation of the intrusion detection system snort performance, Electron. Elect. Eng. 7 (87) (2008) 15–18.
[29] B. Ribeiro, T. Ye, D. Towsley, A resource-minimalist flow size histogram estimator, in: IMC '08, 2008, pp. 285–290.
[30] X. Shi, D.-M. Chiu, J.C.S. Lui, An online framework for catching top spreaders and top scanners. TR, available at <http://personal.ie.cuhk.edu.hk/~sxg007/framework/TR.pdf>.
[31] S. Singh, C. Estan, G. Varghese, S. Savage, Automated worm fingerprinting, in: OSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 4–4.
[32] Snort, <www.snort.org>.
[33] P. Truong, F. Guillemin, Estimating Local Cardinalities in a Multidimensional Multiset, LNCS, 2007, pp. 172–175.
[34] S. Venkataraman, D. Song, P.B. Gibbons, A. Blum, New streaming algorithms for fast detection of superspreaders, in: SNDSS '05, 2005, pp. 149–166.
[35] K.-Y. Whang, B.T. Vander-Zanden, H.M. Taylor, A linear-time probabilistic counting algorithm for database applications, ACM Trans. Database Syst. 15 (2) (1990) 208–229.
[36] H.C. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, J. Xu, A data streaming algorithm for estimating entropies of od flows, in: IMC '07, ACM, New York, NY, USA, 2007, pp. 279–290.
[37] Q. Zhao, A. Kumar, J. Xu, Joint data streaming and sampling techniques for detection of super sources and destinations, in: IMC '05, 2005.
[38] Q.G. Zhao, A. Kumar, J. Wang, J. Xu, Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices, in: SIGMETRICS '05, 2005, pp. 350–361.

Xingang Shi received the B.Sc. degree from Tsinghua University, and is currently a Ph.D. student in the Department of Information Engineering at the Chinese University of Hong Kong. His research interests include network measurement, streaming algorithms and routing protocols.

Dah-Ming Chiu received the B.Sc. degree from Imperial College London and the Ph.D. degree from Harvard University in 1975 and 1980. After twenty years in industry (Bell Labs, DEC and Sun), he is now a professor in the Department of Information Engineering in CUHK. His research interest includes Internet, wireless networks and P2P networking. He is an editor for IEEE/ACM Transactions on Networking, and TPC member for various IEEE conferences including Infocom, ICNP and IWQoS.

John C.S. Lui received his Ph.D. in Computer Science from UCLA. He later joined the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include theoretic/applied topics in data networks, distributed multimedia systems, network security and mathematical optimization and performance evaluation theory. John was the TPC co-chair of ACM Sigmetrics 2005 and the general co-chair of the International Conference on Network Protocols 2006. His personal interests include films and general reading.