Data Streaming Algorithms for Estimating Entropy of Network Traffic

Ashwin Lall
University of Rochester

Vyas Sekar
Carnegie Mellon University

Mitsunori Ogihara*
University of Rochester

Jun (Jim) Xu†
Georgia Institute of Technology

Hui Zhang‡
Carnegie Mellon University

Univ. of Rochester Comp. Sci. Dept. Technical Report TR-2005-886
November 20, 2005

Abstract

Using entropy of traffic distributions has been shown to aid a wide variety of network monitoring applications such as anomaly detection, clustering to reveal interesting patterns, and traffic classification. However, realizing this potential benefit in practice requires accurate algorithms that can operate on high-speed links, with low CPU and memory requirements. Estimating the entropy in a streaming model to enable such fine-grained traffic analysis has been a challenging problem. We give lower bounds for this problem, showing that neither approximation nor randomization alone will let us compute the entropy efficiently.

We present two algorithms for randomly approximating the entropy in a time and space efficient manner, applicable for use on very high speed (greater than OC-48) links. Our first algorithm for entropy estimation, inspired by the seminal work of Alon et al. for estimating frequency moments, has strong theoretical guarantees on the error and resource usage. Our second algorithm utilizes the observation that the efficiency can be substantially enhanced by separating the high-frequency items (or elephants), from the low-frequency items (or mice). Evaluations on real-world traffic traces from different deployment scenarios demonstrate the utility of our approaches.

1 Introduction

In network traffic flow analysis there has been a shift of focus from simple volume-based analysis to network flow distribution-based analysis. Much work has been published for making inference about the network status from such statistics [16, 24, 12]. Intrinsically, distribution-based analysis could capture the network status more succinctly than volume-based analysis would, but it requires appropriate metrics to encapsulate and capture features of the underlying traffic distribution.

*Supported in part by grants Xerox/NYSRAT #C040130 and NSF-EIA-0205061.
†Supported in part by NSF grant NETS-NBD 0519745 and NSF CAREER Award ANI 0238315.
‡Supported in part by grants NSF CNS-0433540 and Army Research Office DAAD19-02-1-0389.

The standard quantities in assessing distributions are the moments (the mean, standard deviation, skewness, kurtosis, etc.). A number of recent empirical studies [24, 16, 23, 7] have suggested the use of *entropy* as a succinct means of summarizing traffic distributions for different applications, in particular, in anomaly detection and in fine-grained traffic analysis and classification. With respect to anomaly detection [16], the use of entropy for tracking changes in traffic distributions provides two significant benefits. First, the use of entropy can increase the sensitivity of detection to uncover anomalous incidents that may not manifest as volume anomalies. Second, using such traffic features provides additional diagnostic information into the nature of the anomalous incidents (e.g., making distinction among worms, DDoS attacks, and scans) that is not available from just volume-based anomaly detection. With respect to fine-grained traffic analysis and traffic classification [24], the entropy of traffic feature distributions offers useful information to measure distance among (traffic) clusters.

While these recent studies demonstrate that using the entropy of traffic distributions has tremendous value for network monitoring applications, realizing the potential benefit requires efficient algorithms for computing the entropy. In general, computing traffic statistics on high-speed links is a hard task, because it is infeasible for traditional methods to keep up with the line-rates, due to constraints on available processing capacity. In addition, constraints imposed on memory make it almost impossible to compute the statistics per flow, or even to maintain per-flow state. Then, the use of sampling comes as a natural solution. Sampling based methods [5, 6] have been shown to be able to reduce the processing and memory requirements, and to be suitable for capturing some traffic statistics. However, it is often the case that one must trade off accuracy for efficiency—the estimates obtained from sampled data may have large errors [10].

One may then naturally wonder whether there are efficient methods for accurately estimating the entropy. In particular, we ask the following questions:

- What amount of resources (time and space) do we provably need to capture the entropy of a stream of packets on a high-speed link?
- Are there efficient algorithms for entropy computation that can operate on high-speed links which have low memory and CPU costs?

To address these questions, data streaming algorithms assume significance. Data streaming algorithms [19] for computing different statistics over input streams have recently received tremendous interest from the networking and theory communities. Data streaming algorithms have the desirable property that both the computational and memory requirements are low. This property makes them ideal for such high-speed monitoring applications. They are also guaranteed to work with *any* distribution, which makes them useful in dealing with data flow for which the distribution is not known.

We thus question whether good streaming algorithms exist for finding the entropy of a stream of objects. The challenge is to design data-streaming-based entropy estimation algorithms that are lightweight in terms of both memory and computational complexity. The primary contribution of this paper is the investigation and application of streaming algorithms to compute the entropy over network traffic streams. We present two algorithms for computing the entropy in a streaming model. The first algorithm is based on the insight that estimating the entropy shares structural similarity with the well-known problem of estimating the frequency moments [1] [1]. Despite the apparent structural similarity, providing theoretical approximation and resource guarantees for entropy estimation

[1]This approach has been independently proposed by Chakrabarti et al. [4]. We will discuss this and other approaches in Section 8, highlighting that while our intellectual trails cross each other on some results, our approaches and evaluations differ substantially in others.

is a challenging task. Our contributions are the identification of appropriate estimator functions for calculating the entropy accurately, and providing proofs of approximation guarantees and resource usage. The theoretical guarantees hold for arbitrary streams, without making any assumptions regarding the underlying distributions and structural properties of their distribution.

Most network traffic data-streams have considerable underlying structure (e.g., they may have a Zipfian or power-law distribution), which suggests that we can optimize algorithms further by leveraging this fact. Our second algorithm builds on the basic streaming algorithm, but can substantially improve the efficiency based on techniques for separating the large (elephant) flows from the small (mice) flows. We use a novel sampling method to bring about this separation.

We evaluate our algorithms on real traffic traces collected from three different deployment scenarios. We observe that the basic algorithm outperforms traditional sampling based approaches, and provides much lower estimation errors while using similar (or lesser) memory resources. We also notice that the observed errors are an order of magnitude smaller than the theoretical error guarantees. While it has proved difficult to provide strong theoretical (i.e., worst-case) guarantees for the second algorithm (which makes use of the elephant-mice separation), we find that the observed errors are substantially lower.

The remainder of the this paper is organized as follows. We introduce the notation that we will use and formally define the problem in Section 2. In Section 3 we prove that any (deterministic) approximation algorithm or (exact) randomized algorithm must use a linear amount of space. Section 4 outlines the basic streaming algorithm, providing theoretical approximation guarantees, while Section 5 provides improvements based on the technique of separating the elephant and mice flows. We evaluate our algorithms on real-world traces in Section 6, confirming the effectiveness of our approaches. We discuss some features of our algorithms in Section 7 and related work in Section 8 before we conclude in Section 9.

2 Problem Formulation

We first outline the notation used in the remainder of the paper, and formulate the problem of estimating entropy in a streaming context.

Throughout this paper we will assume that all items coming over the stream are drawn from the set $[n] = \{1, 2, 3, \ldots, n\}$. For example if we are interested in measuring the entropy of packets over various application ports, then n is the number of ports (maximum of 65535 ports for each protocol). Similarly, if we are interested in measuring the entropy of packets over unique source or destination addresses in the traffic stream, then n would have a maximum value of 2^{32} for 32-bit IPv4 addresses.

We will denote the frequency of item $i \in [n]$ (e.g. the number of packets seen at port i) by m_i and the total number of items in the stream by m, i.e., $m = \sum_{i=1}^{n} m_i$. The ith item observed in the stream will be denoted by $a_i \in [n]$. We define n_0 to be the number of distinct items that are actually present in the stream, since it is possible that not all n items actually appear in the stream. As a simple example consider a stream drawn from a set of $n = 4$ different possible objects $\{A, B, C, D\}$. Let the stream $X = (A, A, B, B, C, A, B, A, C)$. For this stream, the total number of items $m = 4 + 3 + 2 = 9$, with the number of distinct items $n_0 = 3$.

The natural definition of entropy (sometimes referred to as sample entropy) in this setting is the expression $H \equiv -\sum_{i=1}^{n} \frac{m_i}{m} \log\left(\frac{m_i}{m}\right)$. Intuitively, the entropy is a measure of the diversity or randomness of the data coming over the stream. The entropy attains its minimum value of zero when all the items coming over the stream are the same and its maximum value of $\log m$ when all the items in the stream are distinct. Unless otherwise specified, all logarithms in this

paper are to the base 2 and we define $0 \log 0 = 0$. For our example stream X, the entropy $H_X = -(4/9)\log(4/9) - (3/9)\log(3/9) - (2/9)\log(2/9) = 1.53$. Often it is useful to normalize this number to between zero and one so as to compare entropy estimates across different measurement epochs. For this purpose, the relative entropy is defined to be $H/\log m$. In our example, the relative entropy is $1.53/\log 9 = 0.48$.

To compute the entropy,

$$\begin{aligned} H &= -\sum_{i=1}^{n} \frac{m_i}{m} \log\left(\frac{m_i}{m}\right) \\ &= \frac{-1}{m}\left[\sum_i m_i \log m_i - \sum_i m_i \log m\right] \\ &= \log(m) - \frac{1}{m}\sum_i m_i \log m_i, \end{aligned}$$

it suffices to compute $S \equiv \sum_i m_i \log m_i$, since we can keep a count of m exactly with $\log m$ bits. For the remainder of this paper we will concern ourselves with estimating the value S. The measure of accuracy that we use to evaluate our estimates, is a standard notion of relative error, which is defined to be $|S - \tilde{S}|/S$, where $\tilde{S}$ is the estimated value and S the real value. For practical applications in traffic monitoring, we require that the relative error be low (say less than 2-3%), so that the accuracy of applications such as anomaly detection and traffic clustering is not affected.

It is important to note that an accurate estimate of S may not necessarily give an accurate estimate of H. In particular, when H is very small and S is close to its maximum value, a small relative error estimation of S may not correspond to a small relative error estimation of H. Let $\tilde{S}$ be the estimated value of S and $\tilde{H}$ the estimated value of H computed from $\tilde{S}$, i.e., $\tilde{H} = \log(m) - \tilde{S}/m$. Suppose we have an algorithm to compute S with at most ϵ relative error. Then, the relative error for H can be bounded as follows:

$$\begin{aligned} \frac{|H - \tilde{H}|}{H} &= \frac{|\log(m) - S/m - \log(m) + \tilde{S}/m|}{H} \\ &= \frac{|S - \tilde{S}|}{Hm} \\ &\leq \epsilon \frac{S}{Hm}. \end{aligned}$$

Note that the relationship between the relative error in H actually depends on the ratio $\frac{S}{Hm}$, which can theoretically become arbitrarily high (if H is close to zero). However, it is possible to prove that, given some reasonable lower bounds for how small H can get, any algorithm that can give an approximation of S with relative error at most ϵ can be converted to one that gives an approximation of H with relative error $\epsilon' = \Omega(\epsilon)$. Since we know that $S \leq m \log m$, if we assume a lower bound of $\alpha \log m$ for H then the relative error for H is at most $\epsilon' = \epsilon/\alpha$. So, any approximation scheme for S can be converted to one for H if we can assume a lower bound on the entropy. Our evaluations on real-world packet traces (Section 6.2) indicate that the relative error for H turns out to be comparable to that of S.

3 Lower Bounds

In this paper we will present a randomized approximation algorithm for computing the value S of a stream. Before we do this, we would like to answer the first question of how much effort is required to

estimate the entropy of a given traffic distribution. We will demonstrate that any exact randomized algorithm or any deterministic approximation algorithm needs at least linear (in the length of the stream) space.

Theorem 1 *Any randomized streaming algorithm to compute the exact value of S when there are at most m items must use $\Omega(m)$ bits of space.*

Proof: We demonstrate that any randomized algorithm to compute S must use $\Omega(m)$ space by reducing the communication complexity problem of set intersection to it. Using communication complexity is a common way to prove lower bounds for streaming algorithms [1, 18]. We show here how to apply it to the computation of entropy.

In the communication complexity model two parties (typically called Alice and Bob), who have non-overlapping but jointly complete parts of the input, wish to compute some function of the input. The communication complexity of the function at input size n is then the largest number of bits that the parties have to communicate using the *best* protocol to compute the function, for any input of size n. There are no bounds on the computational power of both parties and the only resource being measured is the number of bits communicated.

For the problem of set intersection, Alice and Bob have subsets A and B of $\{1, \ldots, N\}$ as input. The question is then whether the sets A and B have any elements in common. It is known that the deterministic communication complexity of this problem is $\Theta(N)$ [15]. It was shown by Kalyanasundaram and Schnitger in [11] that any communication complexity protocol for set intersection that has probability of error at most δ, for any $\delta < 1/2$, must use $\Omega(N)$ bits of communication. We make use of this result in the proof.

Let us assume that we have a randomized streaming algorithm that computes $\sum_i m_i \log m_i$ for any stream exactly using s bits of space. This gives rise to a communication complexity protocol for computing set intersection that works as follows.

Suppose that Alice and Bob have as input subsets of the set $\{1, \ldots, m/2\}$. Alice enters her input (in any arbitrary order) as a stream into the algorithm and sends a snapshot of the state of her memory (at most s bits) to Bob. Bob then runs the algorithm, starting with that memory configuration, and enters his entire input. At the end of this round, Bob checks the output of the algorithm—if the output is zero, he outputs "disjoint," otherwise he outputs "not disjoint."

The above protocol relies on the fact that any items that have frequency at most one do not count toward the sum S (since $1 \log 1 = 0 \log 0 = 0$). So, the value of S computed is simply twice the size of the intersection. If we find that the intersection has size zero then we know that Alice and Bob's sets are disjoint, otherwise they have something in common.

Hence, even if the streaming algorithm is randomized, it must use $s = \Omega(m)$ bits. If it used fewer bits it would lead to a protocol for set intersection with less than $\Omega(n)$ communication, which we know from [11] to be impossible.

Theorem 2 *Any streaming algorithm to approximate S with relative error less than $1/3$ must use $\Omega(m)$ bits of space.*

Proof: The proof that any (non-randomized) approximation algorithm is inefficient is similar to the proof of Proposition 3.7 in [1]. Let G be a family of $2^{\Omega(m)}$ subsets of $[2m]$, each of cardinality $m/2$ so that any two distinct members of G have at most $m/4$ elements in common (It is possible to show such a G exists using a counting argument.).

Let us assume for a contradiction that there exists a deterministic streaming algorithm that estimates S with relative error at most $1/3$, using sublinear space. For every pair of elements $G_1, G_2 \in G$, let $A(G_1, G_2)$ be the sequence of length m consisting of the elements of G_1 in sorted

order followed by the elements of G_2 in sorted order. By the pigeonhole principle, if the memory used by the algorithm has less than $\log |G| = \Omega(m)$ bits, then at least two distinct subsets $G_i, G_j \in G$ result in the same memory configuration when their contents are entered into the algorithm. Hence, the algorithm cannot distinguish between the streams $A(G_i, G_i)$ and $A(G_j, G_i)$. For the input $A(G_i, G_i)$ we have that $S = (m/2)(2 \log 2) = m$, but for $A(G_j, G_i)$, $S \leq (m/4)(2 \log 2) = m/2$. Now, if the relative error for $A(G_i, G_i)$ is less than $1/3$, its estimated value is more than $2m/3$, but if the relative error for $A(G_j, G_i)$ is less than $1/3$ its estimated value is less than $2m/3$. Therefore, the algorithm makes a relative error of at least $1/3$ on at least one of these inputs. This tells us that any non-randomized algorithm must either use $\Omega(m)$ space or have a relative error of at least $1/3$.

Thus, we see that if we use only randomization or only approximation we cannot hope to use a sublinear amount of space. As a result, the following algorithms that we present are both randomized and approximate. Fortunately, when we allow for these two relaxations we get algorithms that are sublinear (in particular, polylogarithmic) in space and time per item.

4 A Streaming Algorithm

In this section we present our first algorithm and show guarantees on the performance and the size of the memory footprint. The basic algorithm is based on the key insight that estimating S is structurally similar to estimating the frequency moments [1]. The advantage of this technique is that it gives an unbiased estimate of the entropy, with strong theoretical guarantees on the space consumption based upon the desired accuracy of the algorithm. We then show how the assumptions and analysis of the algorithm can be further tightened.

4.1 Algorithm

As demonstrated in the previous section, randomization and approximation alone do not allow us to estimate S efficiently. Hence, we present an algorithm that is an (ϵ, δ)-approximation of S. An (ϵ, δ)-approximation algorithm is one that has a relative error of at most ϵ with probability at least $1-\delta$, i.e., $Pr(|X - \tilde{X}| \leq X\epsilon) \geq 1 - \delta$, where X and $\tilde{X}$ are the real and estimated values, respectively.

This algorithm uses the idea of the celebrated Alon–Matias–Szegedy frequency moment estimation algorithm [1]. We compute the median of $g = \lceil 2 \log (1/\delta) \rceil$ variables, $Y_1, \ldots, Y_g$, each of which is the average of a group of $z = \lceil 32 \log m/\epsilon^2 \rceil$ variables. Each variable is an unbiased estimator of S for the stream and is computed as follows.

For each of the $g * z$ variables, we choose uniformly at random a position in the stream (i.e., an integer $1 \leq p \leq m$). For each such position we keep a counter c for that item from that position on. Note that since we always count the item at position p, we have that $c \geq 1$. The variable is then defined to be

$$X = m(c \log c - (c-1) \log (c-1)).$$

We present the pseudocode for this algorithm in Algorithm 1. The run of the algorithm is divided into the pre-processing, online and post-processing stages. In the pre-processing stage we need to choose $z * g$ locations in the stream. Note that for this stage we need to know the length of the stream to both compute z and to choose the random locations. The choice of the random locations can be deferred as described in [1] and to compute z we can use a safe overestimate for $\log m$ without increasing the space too much. In the online stage we update at most one record per item, using a data structure described in the following section. In the post-processing stage we need to average the variables X and compute the median of the averages.

Algorithm 1: The streaming algorithm

{Pre-processing stage}
$z := \lceil 32 \log m/\epsilon^2 \rceil$, $g := 2 \log (1/\delta)$
choose $z * g$ locations in the stream at random
{Online stage}
for each item a_i in the stream **do**
  if a_i already has one or more counters **then**
    increment all of a_i's counters
  if i is one of the randomly chosen locations **then**
    start keeping a count for a_i, initialized at 1
{Post-processing stage}
{interpret the $g * z$ counts as a matrix c of size $g \times z$}
for $i := 1$ to g **do**
  for $j := 1$ to z **do**
    $X_{i,j} := m * (c_{i,j} \log c_{i,j} - (c_{i,j} - 1) \log (c_{i,j} - 1))$
for $i := 1$ to g **do**
  $avg[i] :=$ the average of the Xs in group i
return the median of $avg[1], \ldots, avg[g]$

4.2 Implementation Details

One major advantage of this algorithm is that it is light weight. For any item in the stream, the algorithm has to update its count if the item is being counted. Checking whether the item is being counted can be done very quickly using a hash table. However, it is possible that a single item has multiple records for it. In the worst case, we would need to update every record for each item. We could greatly improve the efficiency of the algorithm by instead keeping a single record for every unique item. This can be implemented by only updating the most recent record for that item and maintaining a pointer to the next most recent record. When the entire stream has been processed, the counts for the older records can be reconstructed from those of the newer ones.

The record data structure that we suggest is illustrated in Figure 1. Each record in our implementation would require $\sim$ 200 bits because we would need to store the item label ITEM_LABEL ($\sim$ 100 bits), the counter for the item COUNTER (32 bits), a pointer CHAINING_PTR (32 bits) to resolve hash collisions if we use chaining and another pointer PREV_PTR (32 bits) to point to the older records for the item. We use a conservative estimate of 100 bits for each item label, assuming that we would store all 5 main IP packet header fields, i.e., ⟨srcaddr, dstaddr, srcport, dstport, protocol⟩.

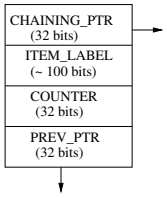


Figure 1: The record data structure

At the end of each epoch the algorithm needs to perform the operations of averaging and finding

the median of a list. However, both these operations only need to be done in the post-processing step. If we make an epoch sufficiently large, then these computations need be done relatively infrequently.

4.3 Theoretical Guarantees

We present analysis that shows we can give strong guarantees while using very little space. The proof is along the lines of the one in [1] and the main contribution here is to show how the variance of the variable X can be bounded to give such a small space requirement. The proof requires the assumption that $S \geq m$ or, equivalently, that $H \leq \log m - 1$. We show in Section 4.5 why this assumption is reasonable.

Theorem 3 *If we assume that $S \geq m$, then Algorithm 1 is an (ϵ, δ)-approximation algorithm for S that uses $O(\log m \log(1/\delta)/\epsilon^2)$ records.*

Proof: We will first show that the variable X is an unbiased estimator for S. We will then make use of Chebyschev's inequality to bound the probability of having a relative error greater than ϵ. Next, we show that if we average $z = \lceil 32 \log m/\epsilon^2 \rceil$ variables, this probability is at most $1/8$. We can then use Chernoff bounds to show that if we take $g = 2\log(1/\delta)$ such averages, with probability at least $1 - \delta$ more than half of them have less than ϵ relative error. In this case, the median of the averages must have relative error less than ϵ.

We first observe that the expected value of each variable X is an unbiased estimate of our desired quantity S:

$$\begin{aligned} E[X] &= \frac{m}{m} \sum_{i=1}^{n} \sum_{j=1}^{m_i} (j \log j - (j-1)\log(j-1)) \\ &= \sum_{i=1}^{n} m_i \log m_i \\ &= S. \end{aligned}$$

To make use of Chebyschev's inequality, we need to bound the variance of X from above, in terms of S^2. The bound proceeds as follows:

$$\begin{aligned} Var(X) &= E(X^2) - E(X)^2 \\ &\leq E(X^2) \\ &= \frac{m^2}{m} \left[\sum_{j=1}^{n} \sum_{i=2}^{m_j} (i \log i - (i-1)\log(i-1))^2 \right]. \end{aligned}$$

Now we observe that

$$\begin{aligned} n \log n - (n-1)\log(n-1) &= \log \frac{n^n}{(n-1)^{n-1}} \\ &\leq \log \frac{n^n}{n^{n-2}} \\ &= 2 \log n, \end{aligned} \tag{1}$$

where the inequality comes from the facts that the logarithm function is monotonically increasing and that for all $n > 1$, $n^{n-2} \leq (n-1)^{n-1}$, which is proven as follows:

For $n = 2$ the fact can easily be checked. For all other n, $n > e$, so

$$\frac{n^{n-2}}{(n-1)^{n-1}} = \frac{1}{n}\left(\frac{n}{n-1}\right)^{n-1} = \frac{1}{n}\left(1 + \frac{1}{n-1}\right)^{n-1}.$$

This is at most $e/n \leq 1$. So, the inequality holds.

Now, substituting (1) into the bound on the variance, we get that

$$\begin{aligned} Var(X) &\leq m\sum_{i=1}^{n}\sum_{j=2}^{m_i}(2\log j)^2 \\ &\leq 4m\sum_{i=1}^{n} m_i \log^2 m_i \\ &\leq 4m\log m\left(\sum_i m_i \log m_i\right) \\ &\leq 4S\log m\left(\sum_i m_i \log m_i\right) \\ &= 4S^2 \log m, \end{aligned}$$

where for the last inequality we make use of our assumption that $S \geq m$.

Let the average of the ith group be Y_i. We know that $Var(Y_i) = Var(X)/z$ and that it is also an unbiased estimator of S. Applying Chebyshev's inequality, we get that for each Y_i,

$$\begin{aligned} Pr(|Y_i - S| > \epsilon S) &\leq \frac{Var(Y_i)}{\epsilon^2 S^2} \\ &\leq \frac{4S^2 \log m}{z\epsilon^2 S^2} \\ &= \frac{4\log m}{z\epsilon^2} \\ &\leq \frac{1}{8}. \end{aligned}$$

Now, by Chernoff bounds we get that with probability at least $1-\delta$, at least $g/2$ of the averages have at most ϵ relative error. Hence, the median of the averages has relative error at most ϵ with probability at least $1 - \delta$.

Note that if we had chosen $z = \lceil \log m/(\epsilon^2 * \delta)\rceil$ we could have guaranteed an error probability of at most δ with just this one bigger group. While the analysis in the proof works well for smaller δ (i.e., $\delta \leq 1/128$), for practical applications we may want to use larger δ. Because of the independence of each run, with $\delta = 10\%$ we detect an attack within one epoch with 90% certainty, within two epochs with 99% certainty and so on. For the case where δ is greater than $1/128 \approx 0.8\%$ we can use the average of a single group of $z = \lceil \log m/(\epsilon^2 * \delta)\rceil$ estimators for our estimate.

The total space (in bits) used by this algorithm is

$$O\Big(\frac{\log m \log(1/\delta)}{\epsilon^2}(\log n + \log m)\Big).$$

For fixed δ and ϵ this algorithm uses $O(\log m)$ records of size $O(\log m + \log n)$ bits.

Numerical Illustration: To put this into a practical perspective, let us consider an example where we have a stream of length $m = 2^{26} \approx 67$ million, with $n_0 = 6$ million distinct items. To compute the entropy exactly, we could have to maintain counts for each item using 6 million item labels and counters (132 bits/record $\times$ 6 million records = 94 MB). Using Algorithm 1 we could approximate the entropy with at most 25% relative error at least 75% of the time with 54 thousand records or 1.4 MB, using 200 bit records as discussed earlier.

4.4 Exact Space Bounds

In practical settings we want know the exact values of the parameters of the above algorithm so that we use as little space as possible. We tighten the bound on the number of groups needed by making the observation that $\frac{j^j}{(j-1)^{j-1}} = j(1+\frac{1}{j-1})^{j-1} < ej$. Here is a tighter (non-asymptotic) analysis for the bound on the variance:

Theorem 4 *If we assume that $S \geq m$, then Algorithm 1 can be modified to use exactly $\lceil \frac{(16 \log m + 64) \log(1/\delta)}{\epsilon^2} \rceil$ records.*

Proof:

$$
\begin{aligned}
E(X^2) &= m \sum_{i=1}^{n} \sum_{j=1}^{m_i} (j \log j - (j-1) \log (j-1))^2 \\
&\leq m \sum_{i=1}^{n} \sum_{j=1}^{m_i} \log^2 (ej) \\
&= m \left(\sum_{i=1}^{n} \sum_{j=1}^{m_i} \log^2 j + m \log^2 e + 2 \log e \sum_{i=1}^{n} \sum_{j=1}^{m_i} \log j \right) \\
&\leq m \left(\sum_{i=1}^{n} m_i \log^2 m_i + m \log^2 e + 2S \log e \right) \\
&\leq S \left(\sum_{i=1}^{n} m_i \log^2 m_i + m \log^2 e + 2S \log e \right) \qquad (2) \\
&\leq S \left(S \log m + m \log^2 e + 2S \log e \right) \\
&= S^2 (\log m + m \log^2 e / S + 2 \log e) \\
&\leq S^2 (\log m + \log^2 e + 2 \log e) \qquad (3) \\
&\leq S^2 (\log m + 5),
\end{aligned}
$$

where (2) and (3) require the assumption that $S \geq m$.

Hence we have that the variance

$$Var(X) = E(X^2) - (E(X))^2 \leq S^2 (\log m + 4).$$

So, we see that $z = \lceil \frac{8 \log m + 32}{\epsilon^2} \rceil$ suffices.

Numerical Illustration: Returning to our example of a stream of size 67 million, the above improvements would drop the number of records for the case of at most 25% error with 75% probability to just 16 thousand (400 KB).

4.5 A Note on Assumptions

For the above analysis, we needed to make the assumption that $S \geq m$. It is not hard to see (and prove) that we need some kind of lower bound on the value of S to protect ourselves from the case that we are trying to distinguish two streams of low S value. If one stream has all unique elements (so that $S = 0$) and another has only one repeated element, then it is very hard to distinguish them. However, we must distinguish them to have less than 100% relative error.

Assuming that $S \geq m$, or that $H \leq \log m - 1$, is reasonable because H attains its maximum value at $\log m$. We now show some other conditions that give us that $S \geq m$, thereby making them reasonable assumptions to make.

Theorem 5 *If $m \geq 2n_0$ then $S \geq m$.*

Proof: It is easy to show using Lagrange multipliers that S attains its minimum value when all the items in the stream have the same count. Hence, a lower bound for S is

$$S \geq \sum_{i=1}^{n_0} \frac{m}{n_0} \log(m/n_0) = m \log(m/n_0).$$

Since we have assumed that $m \geq 2n_0$, this gives us that $S \geq m \log(m/n_0) \geq m \log 2 = m$.

Hence, we need only assume that each item in the stream appears at least twice on average. This assumption protects us from the case described earlier and in any setting where S can get arbitrarily small. We feel that in any practical setting this simple assumption is very reasonable. For example, on all the traces that we experimented on, the factor m/n_0 was in the range of 50 to 300.

4.6 A Constant-space Solution

As it turns out, if we make a stronger (but still reasonable) assumption on how large the entropy can get, we can make the space usage of the algorithm *independent of* m (assuming fixed sized records). Upper bounding the entropy is reasonable to do since even during abnormal events (e.g worm attacks), when the randomness of the distributions are increased, there will still be a sufficiently large amount of legitimate activity to offset the increased randomness.

Recall that H attains its maximum at $\log m$, when each of the m items in the stream appears exactly once. We will assume that $H \leq \beta \log m$. This gives us the following bound on S:

$$\begin{aligned} S &= m \log m - mH \\ &\geq m \log m - \beta(m \log m) \\ &= (1 - \beta) m \log m. \end{aligned}$$

We can now apply this to decrease the space usage of our algorithm:

Theorem 6 *If we assume that $H \leq \beta \log m$, then Algorithm 1 can be modified to use exactly $\lceil \frac{64 \log(1/\delta)}{(1-\beta)\epsilon^2} \rceil$ records.*

Proof: We once again bound the variance:

$$\begin{aligned} Var(X) &= E(X^2) - E(X)^2 \\ &\leq E(X^2) \\ &= \frac{m^2}{m}\left[\sum_{j=1}^{n}\sum_{i=2}^{m_j}(i\log i-(i-1)\log(i-1))^2\right] \\ &\leq m\sum_{i=1}^{n}\sum_{j=2}^{m_i}(2\log j)^2 \\ &\leq 4m\sum_{i=1}^{n} m_i \log^2 m_i \\ &\leq 4m\log m\left(\sum_i m_i \log m_i\right) \\ &\leq 4S^2/(1-\beta). \end{aligned}$$

Hence, we need only $z = \frac{32}{(1-\beta)\epsilon^2}$ groups, which is independent of m. The desired bound on the number of records follows from this.

Numerical Illustration: For a stream with 67 million packets, if we make the simple assumption that the entropy never goes above 90% of its maximum value then we need 21 thousand records (525 KB), and if we assume that it never exceeds 75% of its maximum value then we only need $8,200$ records (205 KB). Note that these space bounds will not increase with the size of the stream—they depend only on the error parameters. Hence, we can use a few hundred kilobytes for arbitrarily large streams, as long as we can safely make an assumption about how large its relative entropy can get.

5 Separating the Elephants from the Mice

The algorithm described in the previous section provides worst-case theoretical guarantees independent of the structure of the underlying traffic distributions. In practice, however, most network traffic streams have significant structure. In particular a simple but useful insight [6] is that traffic distributions often have a clear demarcation between large flows (or *elephants*), and smaller flows (or *mice*). A small number of elephant flows contribute a large volume of traffic, and for many traffic monitoring applications it may often suffice to estimate the elephants accurately.

In our second algorithm (see Algorithm 2) we make use of the idea of separating the elephants from the mice in the stream. By separately estimating the contribution of the elephants and mice to the entropy we can further improve the accuracy of our results. This method can be shown to be very powerful in increasing the accuracy and decreasing the space usage of the algorithm. We believe that such a *sieving* idea has much broader applicability. Other streaming algorithms for estimating different traffic statistics can potentially benefit by using such an idea.

By sieving out the elephants, or high-frequency items, from the stream, the algorithm will be more space-efficient. Intuitively, the amount of space needed by the first algorithm is directly proportional to the variance of the estimator X (see Section 4.3), and so if we remove the high-count items we will significantly decrease the variance of the estimator and hence the space required.

For this algorithm we change the method of sampling slightly. Rather than pre-compute positions in the stream (which requires foreknowledge of the length of the stream), we sample each location

Algorithm 2: The sieving algorithm

```
{Online stage}
for each item in the stream do
  if the item is sampled then
    if the item is already being counted then
      promote the item to elephant status
    else
      allocate space for a counter for this item
  else
    increment the counter for this item, if there is one
{Post-processing stage}
S_e := 0
for each item marked as an elephant (with estimated count c) do
  S_e := S_e + c log c
estimate the contribution of the mice S_m from the remaining counts using Algorithm 1
return S_e + S_m
```

with some small probability. If an item is sampled *exactly once*, then we consider it a mouse and compute the entropy of the mice using the previous algorithm. If an item is sampled *more than once*, we consider it an elephant and estimate its exact value. This method of sampling is similar to the Sample and Hold algorithm described in [6]—after we sample an item, we keep an exact count for it from then on. In fact, we follow the suggestion of the paper in estimating the elephants by adding $1/p$, the expected number of instances of the item that we skip before we start sampling it. Note that this method is different from [9] in that we are looking for items that are sampled multiple times, not necessarily in consecutive samples.

The record data structure for this sampling method is similar that used by Algorithm 1. The main difference is that we no longer need a pointer to older copies of an item since we only maintain a single count for each unique item. To be able to tell whether the item has been sampled before or not (to determine whether it should be promoted to an elephant) we require just a single additional bit. Thus, we see that this sampling method requires minimal overhead to separate the elephants from the mice.

The sieving algorithm assumes that every flow that is sampled twice is "elevated" to the status of an elephant. Rather than choose the elevation threshold, we evaluated different values of the threshold before we arrive at the number two. Figure 2 shows the relative error as a function of k, the threshold for promoting mice to elephants. We use the packet trace *Trace 1*. The next section provides further details on the traces used in our evaluations. We observe that the lowest error is achieved with a value of $k = 2$, and the error monotonically increases with k. Intuitively, with a higher strike-threshold, we decrease the number of elephants, and we do not achieve the desired elephant-mice separation.

A natural question with such a sieving algorithm is one regarding the relative weights of the two different contributing factors. Intuitively, if either the elephant or the mice flows are not substantial contributors, then we can potentially reduce the space usage further by ignoring the contribution of the insignificant one. We empirically confirmed the need for accurate estimation of both the elephant and the mice flows. Figure 3 shows the relative contribution of the elephant and mice flows to the S estimate. We observe that both elephant and mice flows have substantial contributions to

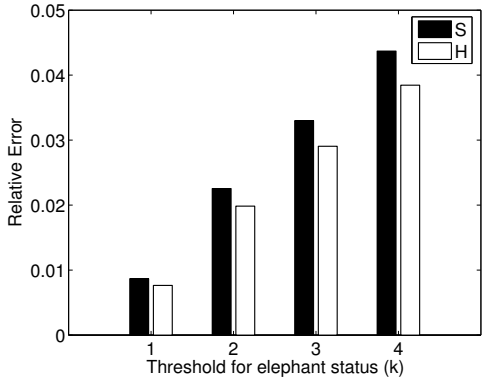


Figure 2: Selecting the threshold k for Sieving

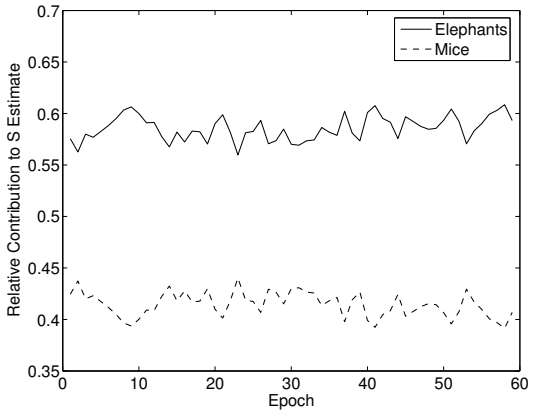


Figure 3: Confirming that estimating both elephants and mice is necessary

the overall estimation, and ignoring one of them can yield inaccurate results for estimating S, and hence H.

6 Evaluation

We first describe the datasets used in this paper. We then present a comparison of the two streaming algorithms introduced in this paper with other sampling based approaches. There are two natural metrics for characterizing the performance of the streaming algorithm for entropy computation: *resource usage* and *error*. The resource usage is related to the number of counters used by different algorithms, which directly translates into the total memory (SRAM) requirements of the algorithm, and the total CPU usage. For the following evaluations, we use the notion of relative error to determine the accuracy of different algorithms.

Datasets: We use three different packet-header traces for evaluating the accuracy of our algorithms. We provide a brief description of each [2].

- University Trace (*Trace 1*): Our first packet trace is an hour-long packet trace collected from USC's Los Nettos collecting facility on Feb 2, 2004. We bin the trace into 1-minute epochs, with each epoch containing roughly 1.7 million TCP packets, 30267 distinct IP addresses, and 15165 ports per minute. We refer to this as *Trace 1* in the following discussion.
- Department Trace (*Trace 2*): We use a 3-hour long packet trace collected on Aug 5, 2003 at the gateway router of a medium sized department with approximately 1350 hosts. We observe all traffic to and from the 1350 hosts behind the access router to the commercial Internet, other non-department university hosts and servers. We bin the dataset into 5-minute epochs for our evaluation, with each epoch observing TCP 500000 packets, 2587 distinct addresses, and 4672 distinct ports on average. We refer to this as *Trace 2* in the following discussion.
- University Trace (*Trace 3*): The third trace we use is an hour-long trace collected at the access link of the university to the rest of the Internet, at UNC on Apr 24, 2003. We bin this trace into 1-minute epochs as with *Trace 1*. Each epoch contains on average 2.5 million packets, 25565 distinct IP addresses, and 8080 unique application ports. We refer to this as *Trace 3*.

Distributions of Interest: We focus on two main types of distributions for our evaluation. The number of distinct source and destination addresses observed in a dataset, and the distribution of traffic across destinations are typically affected by network attacks, including DDoS and worm attacks. We track the distribution of traffic across different addresses for the source and destination addresses. Understanding the application mix that traverses a network can usually be mapped into a study of the distribution of traffic on different application ports. The distribution of traffic across different ports can also be indicative of scanning attacks or the emergence of new popular applications. In each case we are interested in the distribution of the number of packets observed at each port or address (source or destination) within the measurement epoch. Lakhina et al. [16] give an overview of different types of network events and distributions that each would affect.

For Algorithm 1 we use the weakest assumption for all our experiments, i.e., that $m/n_0 \geq 2$. To confirm that this assumptions holds for our traces and distributions, we present the ratio m/n_0 for them here. For *Trace 1* the ratio is roughly 55 for the addresses and 115 for the ports. For *Trace 2*, $\frac{m}{n_0}$ is around 193 for the addresses and 95 for the ports. Lastly, the ratio is around 97 for the addresses and 300 for the ports in *Trace 3*. Thus we see that in all of our traces the assumption is satisfied.

6.1 Comparison with Sampling Algorithms

We first evaluate the accuracy of estimation of our streaming algorithms by comparing them against the following:

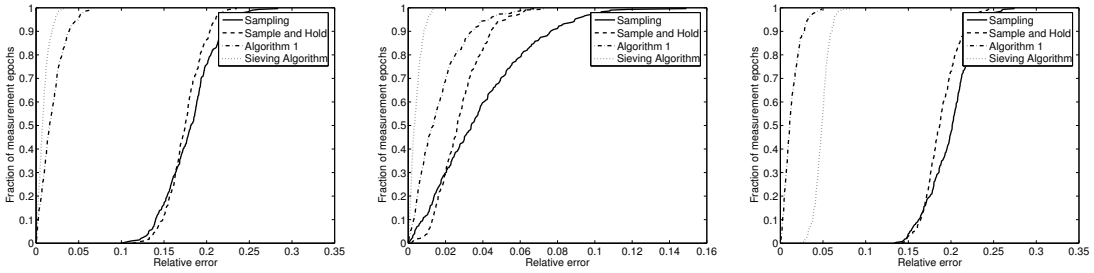
[2]The university traces are available on request from the respective universities. The department trace is a private dataset, we do not provide details on where it was collected to preserve anonymity.

1. **Sampling:** This is the well-known uniform packet sampling approach used in most commercial router implementations [20]. Given a sampling probability p, the sampling approach will pick each packet independently with probability p. The estimation of S and H is performed over the set of sampled packets, after normalizing the counts by $1/p$.
2. **Sample and Hold:** This is the sampling approach proposed by Estan and Varghese [6]. Here given a sampling probability p, the algorithm picks each item in the stream with probability p and and keeps an exact count for that item from that point on. Each sample is incremented by $1/p$ to account for occurrences of the item before it was sampled.

The Sieving algorithm introduced in Section 5 is also conceptually a sampling algorithm, similar to *Sample and Hold* which selects a sampling probability p apriori. We want to perform a fair comparison of the performance of such approaches that use the sampling probability to determine the number of records to keep track of. Therefore, we pick the (ϵ, δ) values for *Algorithm 1*, such that the number of counters used is the same. For *Trace 1* we select p, and (ϵ, δ) values so that the number of records to keep track of is roughly 1000. Note that the total memory usage with 1000 records is only $1000 * 200$ bits $= 24$ KB, assuming that each record requires 200 bits as discussed earlier. It is evident the memory footprint of our algorithms with such parameters (i.e the total SRAM usage), is small enough to allow these to operate on very-high speed routers (greater than OC-192 interfaces). Similarly for *Trace 3* we choose the same p as that chosen for *Trace 2*, and this yields roughly 1700 records ($\sim$ 40KB of SRAM) for each of the algorithms For *Trace 2* we select the parameters so that the number of records that each algorithm tracks is roughly 250 ($\sim$ 6KB of SRAM). Since the number of distinct items observed in the smaller deployment scenario is much smaller, we choose a smaller sampling rate.

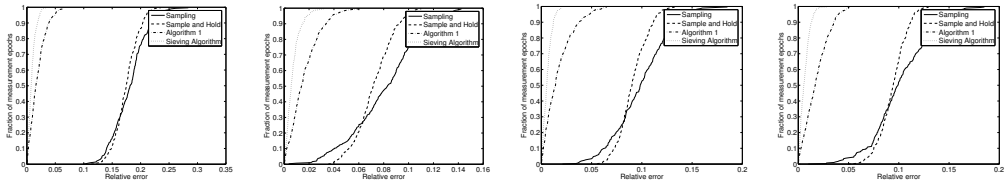
We implemented and tested our algorithms on commodity hardware (Intel Xeon 3.0 GHz desktops with 1 GB of RAM). We found that the total CPU utilization for the streaming algorithms was very low—even though we used a preliminary implementation with very few code optimizations, each measurement epoch took less than 10 seconds to process when the epoch length was an entire minute. This demonstrates that our algorithm can comfortably run in real-time. We also found that the post-processing step consumed a negligible fraction of the time of each run.

Since all of the algorithms are randomized or sampling-based, for the following results we present the mean relative errors and estimates over 5 independent runs. We found that the standard deviations were very small, and do not present the deviations for clarity of presentation.



(a) Trace 1 (b) Trace 2 (c) Trace 3

Figure 4: Comparing performance of different traces for estimating destination address entropy



(a) Destination Address (b) Source Address (c) Destination Port (d) Source Port

Figure 5: Comparing different distributions, using *Trace 1*

Figure 4 compares the performance of different algorithms across different traces. We use the relative error in estimating the entropy of destination addresses in each case. The figures show the CDF of the relative error observed across different measurement epochs. We observe that the streaming algorithms consistently outperform the sampling based approaches. For example, on *Trace 1* we observe that the worst-case relative error with the sampling based approaches can be as high as 25%, whereas the streaming algorithms guarantee a error of at most 6% (Algorithm 1) and 4% (Sieving). We also find that the sieving algorithm provides substantially more accurate estimates for the same space usage compared to the basic streaming algorithm, even though the theoretical guarantees hold only for the basic algorithm. In case of *Trace 2*, which is a substantially smaller trace from a smaller deployment scenario, we observe that the gap between the basic streaming algorithm and the sampling approaches is less pronounced. The reason is that the smaller deployment scenario observes fewer distinct addresses on average, and the accuracy of the streaming algorithms is not substantially different from the pure sampling based approaches. Even in this case, we observe that the sieving algorithm has a worst-case error of at most 1%, which bodes well for the practical utility of the streaming algorithms for traffic monitoring applications.

Interestingly, we find that in *Trace 3* Algorithm 1 gives better performance than the sieving algorithm, whereas in the other two traces the sieving algorithm substantially outperforms the basic streaming algorithm. To investigate this discrepancy we show in Figure 6, the relative contribution of the elephant and mice flows to the overall estimate of S in *Trace 3*. While we observed earlier in Section 5, the contribution of the elephants was higher, we observe in this trace that the contribution of the mice is much higher. In such scenarios, it is therefore likely that Algorithm 1 may give more accurate estimates than the sieving algorithm.

For the rest of the discussion, for brevity we only present the results from *Trace 1*, and summarize the results from *Trace 2* and *Trace 3*. Figure 5 compares the CDF of relative error across measurement epochs, for different distributions of interest from *Trace 1*. We observe a similar trend across algorithms: the sieving algorithm is consistently better than Algorithm 1, which again is substantially more accurate than the sampling based approaches.

Both the streaming algorithms have a worst-case error of 3% and mean error of less than 1% across all the different traffic metrics of interest, which appears a tolerable operating range for typical monitoring applications, confirming the practical utility of our approaches. We summarize the results for the other two traces in Table 1 and Table 2.

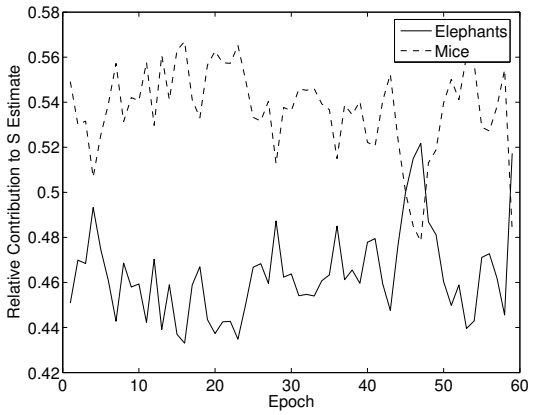


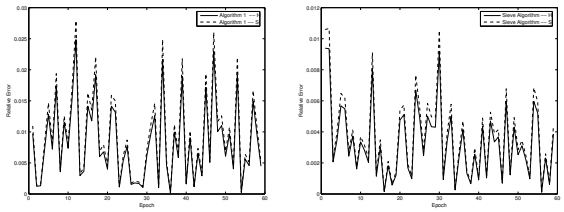
Figure 6: Relative contribution of elephant and mice flows in Trace 3

Table 1: Trace 2: Mean relative error in S estimate

Distribution	Sample	Sample&Hold	Algo. 1	Sieving
DSTADDR	0.038	0.027	0.015	0.004
SRCADDR	0.037	0.026	0.013	0.003
DSTPORT	0.046	0.040	0.017	0.005
SRCPORT	0.048	0.041	0.016	0.004

6.2 Error in estimating entropy

Recall from our discussion in Section 2, that it may be the case an accurate estimation of S does not necessarily translate into an accurate estimate of H. However, we find from our evaluations that the streaming algorithms can yield very accurate estimates of H as well. Figures 7(a) and 7(b) compare the relative error in estimating S to the relative error in estimating H, for Algorithm 1 and the sieving algorithm respectively. We observe that across different traces and distributions, that the relative error in estimating H is very low as well (less than 2% error).



(a) Algorithm 1 (b) Sieving algorithm

Figure 7: Relative error in S vs. relative error in H

Figure 8 also provides visual confirmation of the utility of the different algorithms, in tracking

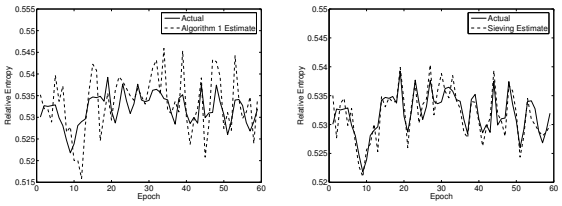
Table 2: Trace 3: Mean relative error in S estimate

Distribution	Sample	Sample&Hold	Algo. 1	Sieving
DSTADDR	0.198	0.187	0.013	0.048
SRCADDR	0.054	0.049	0.016	0.012
DSTPORT	0.116	0.108	0.017	0.037
SRCPORT	0.067	0.062	0.016	0.017

Table 3: Trace 2: Mean relative error in H estimate

Distribution	Sample	Sample&Hold	Algo. 1	Sieving
DSTADDR	0.079	0.057	0.033	0.008
SRCADDR	0.078	0.054	0.028	0.007
DSTPORT	0.089	0.077	0.033	0.009
SRCPORT	0.094	0.081	0.032	0.009

the relative entropy (H_{rel}) for destination addresses. The sieving algorithm once again appears to have greatest accuracy, which can be confirmed with visual inspection. We summarize the results for the other two traces in Table 3 and Table 4, and we observe that in each case the error in H is comparable (or less than) the corresponding error in S (Tables 1 and 2 respectively).



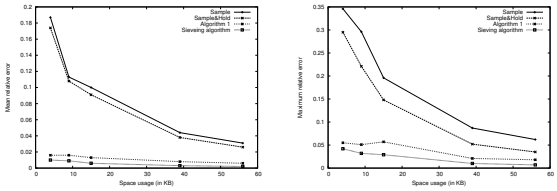
(a) Algorithm 1 (b) Sieving algorithm

Figure 8: Verifying the accuracy in estimating the relative entropy

Last, we vary the memory consumption of the algorithm, and show how the mean and maximum relative errors (for destination address entropy on *Trace 1*) vary as a function of the memory usage in Figure 9. We observe that the streaming algorithms have an order of magnitude lower error than the sampling algorithms, and can achieve very high accuracy ($< 1\%$ mean error), even with as low as 9 KB of SRAM usage. Note that even though the sampling algorithms also can give reasonably low errors at higher memory consumption (> 30 KB), the corresponding sampling rates are much higher (> 1 in 200 packet sampling) than that is feasible for very high-speed links.

Table 4: Trace 3: Mean relative error in H estimate

Distribution	Sample	Sample&Hold	Algo. 1	Sieving
DSTADDR	0.147	0.138	0.009	0.035
SRCADDR	0.081	0.073	0.024	0.017
DSTPORT	0.122	0.114	0.018	0.039
SRCPORT	0.093	0.086	0.023	0.024



(a) Mean error (b) Maximum error

Figure 9: Relative error in estimating H vs. memory usage for different algorithms

7 Discussion

One natural question that arises out of our evaluations is the substantial gap between the theoretical error guarantees and the errors observed in practice. Specifically, for Algorithm 1 we observe that the empirical error is at least one order of magnitude smaller than the error guarantee. This behavior can be explained by the nature of the guarantee given by the algorithm. It must guarantee the error bound for any stream with *any* distribution. Real-world packet traces have considerable underlying structure that the algorithm cannot directly take advantage of.

It now follows that one way to tighten the bounds on the space/error tradeoff is to make reasonable assumptions about the distribution of the stream and have our algorithms take advantage of them. In Section 4.6 we demonstrate this by making the simple assumption that the relative entropy of the stream never goes above some fixed constant. This gives us an algorithm that needs a fixed number of records, independent of the size of the stream. Reasonable assumptions like these can help in tightening our space bounds. However, we did not want to make assumptions that were so strong so that these algorithms could only be used in very specific contexts.

It is a common observation that network packets have a skewed Zipfian distribution. We took advantage of this fact by separating out the few high-count elephants to facilitate the estimation of the remainder more accurately. In doing so, however, we do not make any assumption about the nature of the stream. Algorithm 2 has the property that if there are no elephants in the stream, then it should perform comparably to Algorithm 1. Hence, we expect that, in general, Algorithm 2 should perform better for highly-skewed distributions, but no better than Algorithm 1 when the skew is less pronounced.

8 Related Work

Many of today's networking monitoring applications use the traffic volume, in terms of flow, packet, and byte counts as the primary metric of choice. These are especially of interest for anomaly detection mechanisms to flag incidents of interest. Some of the well-known methods include signal analysis [2], forecasting [3, 21], and other statistical approaches [25, 17].

There has been a recent interest in using entropy and traffic distribution features for different network monitoring applications. Lakhina et al. [16] use the entropy to augment anomaly detection and network diagnosis, within their PCA framework. Others have suggested the use of such information measures for tracking malicious network activity [7, 23]. Xu et al. [24] use the entropy as a metric to automatically cluster traffic, to infer patterns of interesting activity. For detecting specific types of attacks, researchers have suggested the use of entropy of different traffic features for worm [23] and DDoS detection [7].

Streaming algorithms have received a lot of interest in the algorithms and networking community. The seminal work is that of Alon et al. [1] who provide a framework for estimating frequency moments. Since then, there has been a huge body of literature produced on streaming algorithms, and this is well surveyed in [19]. Kumar et al. use a combination of counting algorithms and Bayesian estimation for accurate estimation of flow size distributions [13, 14]. Streaming algorithms have also been used for identifying heavy-hitters in streams [22, 26].

While the entropy can theoretically be estimated from the flow size distribution [13], computing the flow size distribution conceptually provides much greater functionality than that required for an accurate estimate of the entropy. The complexity of estimating the flow size distribution is significantly higher than the complexity of estimating the entropy, requiring significantly more memory and effort in post-processing.

We are aware of two concurrent efforts in the streaming algorithms community for estimating entropy. Chakrabarti et al. [4] independently proposed an algorithm to estimate S that is similar to Algorithm 1. In this paper we show how this algorithm can be modified to use a number of records that is independent of the size of the stream if we make a simple assumption on how large the relative entropy can get. McGregor et al. [8] outline algorithms estimating entropy and other information-theoretic measures in the streaming context. Their algorithm for estimating the entropy of a stream makes assumptions about H (while we only make assumptions about the relative entropy) and their estimates are always biased. Our algorithms provide unbiased estimates of the entropy, unlike their approach, and do not make strong assumptions regarding the underlying distribution. Importantly, in contrast to these purely theoretical approaches, our algorithms are designed taking practical implementation concerns and constraints into consideration, while still providing strong theoretical guarantees. We provide extensive empirical validation of the utility and accuracy of our algorithms on real datasets. We also note that our novel sieving algorithm outperforms Algorithm 1.

9 Conclusions

In this paper, we addressed the need for efficient algorithms for estimating the entropy of network traffic streams, for enabling several real-time traffic monitoring capabilities. We presented lower bounds for the problem of estimating the entropy of a stream, demonstrating that for space-efficient estimation of entropy, both randomization and approximation are necessary. We provide two streaming algorithms for the problem of estimating the entropy. The first algorithm is based on the key insight that the problem shares structural similarity with the problem of estimating frequency moments over streams. By virtue of the strong bounds that we obtain on the variance of the estimator variable, we are able to limit the space usage of the algorithm to polylogarithmic in the length of

the stream. Under some practical assumptions of the size of the entropy, we also give an algorithm that samples a number of flows that is independent of the length of the stream. We also identified a method for increasing the accuracy of entropy estimation by separating the elephants from the mice. Our evaluations on multiple packet-traces demonstrate that our techniques produce very accurate estimates, with very low CPU and memory requirements, making them suitable for deployment on routers with multi-gigabit per second links.

Acknowledgments

We would like to thank A. Chakrabarti, K. Do Ba, and S. Muthukrishnan for their useful discussion and for kindly sharing the most recent version of their paper [4] with us.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of ACM STOC*, 1996.

[2] P. Barford, J. Kline, D. Plonka, and A. Ron. A Signal Analysis of Network Traffic Anomalies. In *ACM SIGCOMM IMW*, 2002.

[3] J. D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proc. of USENIX LISA*, 2000.

[4] A. Chakrabarti, K. Do Ba, and S. Muthukrishnan. Estimating entropy and entropy norm on data streams. Technical Report DIMACS TR 2005-33, DIMACS.

[5] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM*, 2003.

[6] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. of ACM SIGCOMM*, 2002.

[7] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to DDoS attack detection and response. In *Proc. of the DARPA Information Survivability Conference and Exposition*, 2003.

[8] S. Guha, A. McGregor, and S. Venkatasubramanian. Streaming and sublinear approximation of entropy and information distances. to appear in Proc. ACM SODA, 2006.

[9] F. Hao, M. Kodialam, and T. V. Lakshman. ACCEL-RATE: a faster mechanism for memory efficient per-flow traffic estimation. In *Proc. of ACM SIGMETRICS*, 2004.

[10] N. Hohn and D. Veitch. Inverting sampled traffic. In *Proc. of ACM/USENIX IMC*, 2003.

[11] B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discret. Math.*, 5(4):545–557, 1992.

[12] V. Karamcheti, D. Geiger, Z. Kedem, and S. Muthukrishnan. Detecting malicious network traffic using inverse distributions of packet contents. In *Proc. of ACM SIGCOMM MineNet*, 2005.

[13] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow distribution. In *Proc. of ACM Sigmetrics/IFIP WG 7.3 Performance*, 2004.

[14] A. Kumar, M. Sung, J. Xu, and E. Zegura. A data streaming algorithm for estimating subpopulation flow size distribution. In *Proc. of ACM Sigmetrics*, 2005.

[15] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, New York, NY, USA, 1997.

[16] A. Lakhina, M. Crovella, , and C. Diot. Mining anomalies using traffic feature distributions. In *Proc. of ACM SIGCOMM*, 2005.

[17] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. of ACM SIGCOMM*, 2004.

[18] K. Levchenko, R. Paturi, and G. Varghese. On the difficulty of scalably detecting network attacks. In *Proc. of ACM CCS*, 2004.

[19] S. Muthukrishnan. Data streams: algorithms and applications. `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[20] Cisco Netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[21] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in measuring internet backbone traffic variability: Models, metrics, measurements and meaning. In *Proc. of International Teletraffic Congress (ITC)*, 2003.

[22] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders . In *Proc. of ISOC NDSS*, 2005.

[23] A. Wagner and B. Plattner. Entropy Based Worm and Anomaly Detection in Fast IP Networks. In *Proc. of IEEE International Workshop on Enabling Technologies, Infrastructures for Collaborative Enterprises*, 2005.

[24] K. Xu, Z.-L. Zhang, and S. Bhattacharya. Profiling internet backbone traffic: Behavior models and applications. In *Proc. of ACM SIGCOMM*, 2005.

[25] Y. Zhang, Z. Ge, M. Roughan, and A. Greenberg. Network anomography. In *Proc. of ACM/USENIX IMC*, 2005.

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