Building a Better Mousetrap

Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, Vijay Vazirani
College of Computing, Georgia Tech

ABSTRACT

Routers in the network core are unable to maintain detailed statistics for every packet; thus, traffic statistics are often based on *packet sampling*, which reduces accuracy. Because tracking large ("heavy-hitter") traffic flows is important both for pricing and for traffic engineering, much attention has focused on maintaining accurate statistics for such flows, often at the expense of small-volume flows. Eradicating these smaller flows makes it difficult to observe communication *structure*, which is sometimes more important than maintaining statistics about flow *sizes*.

This paper presents *FlexSample*, a sampling framework that allows network operators to get the best of both worlds: For a fixed sampling budget, FlexSample can capture significantly more small-volume flows for only a small increase in relative error of large traffic flows. FlexSample uses a fast, lightweight counter array that provides a coarse estimate of the size ("class") of each traffic flow; a router then can sample at different rates according to the class of the traffic using *any* existing sampling strategy. Given a fixed sampling rate and a target fraction of sampled packets to allocate across traffic classes, FlexSample computes packet sampling rates for each class that achieve these allocations *online*. Through analysis and trace-based experiments, we find that FlexSample captures at least 50% more mouse flows than strategies that do not perform class-dependent packet sampling. We also show how FlexSample can be used to capture unique flows for specific applications.

1. Introduction

Monitoring high-speed network links has traditionally focused on maintaining accurate statistics for large-volume flows, which are useful for billing and traffic engineering. Routers in the network core face strict resource constraints when collecting traffic statistics on networks with high-speed links. Thus, these routers must aggressively sample packets, potentially throwing away a large amount of information. Much work has focused on recovering accurate statistics for large traffic flows ("heavy hitters" or "elephants"), but almost no attention has been devoted to recovering low-volume flows ("mice"). As the needs of operators evolve, it is becoming clear that focusing on capturing statistics about low-volume flows can be useful for traffic classification, application identification, forensics, and detection of low-volume threats (*e.g.*, botnets). Many of these monitoring applications do not require accurately tracking traffic volumes, but rather traffic *structure* [17].

Recovering the structure of network flows, which we call the *communication graph*, (*i.e.*, "Who is talking to whom?") can be useful for detecting application traffic from flow statistics alone. Unfortunately, conventional random sampling (*e.g.*, [16, 23]) captures packets primarily from large traffic flows, and existing extensions to conventional sampling focus on recovering accurate statistics for the relatively small number of heavy hitters. Recovering the communication graph, however, requires collecting as many unique flows as possible, and existing schemes are not equipped to do this. Collecting these edges while maintaining a low overall uniform sampling rate is difficult: naïve coupon collection requires $O(n \log n)$ trials to recover packets from n flows.

Sampling must become more flexible as the needs of network operators diversify. This paper presents a new traffic monitoring framework, *FlexSample*, that allows network operators to get the best of both worlds: It allows operators to capture disproportionately more small flows—likely enough for identification and classification of traffic flows based on the structure of the traffic—for a moderate tradeoff in the relative accuracy of flow size estimates on the captured flows.

1.1 Main Idea: A Two-Stage Approach

The intuition behind FlexSample is quite simple: It uses a fast, coarse-grained counter (or classifier) that can maintain statistics for every packet to assist traffic monitoring techniques that are more resource constrained. In other words, the classifier provides hints about interesting traffic that should be monitored more closely. In this paper, we apply FlexSample to achieve a specific goal: capturing the structure of network traffic as completely as possible.

As shown in Figure 1, FlexSample comprises two distinct stages: complete (but coarse) **counting** and selective (but detailed) **sampling**. Prior to running FlexSample, an operator specifies the fraction of packets to be captured from each size range of a traffic distribution (*e.g.*, $1/3$ of sampled packets should come from "elephant" flows and the rest should come from "mouse" flows). For a given sampling "budget", an operator might choose to allocate more of this budget to mice to capture more of the communication graph for a modest increase in flow size estimation error. For example, an router could disproportionately sample low-volume flows, which would result in more unique flows being captured.

Georgia Tech CSS Technical Report GIT-CSS-07-01

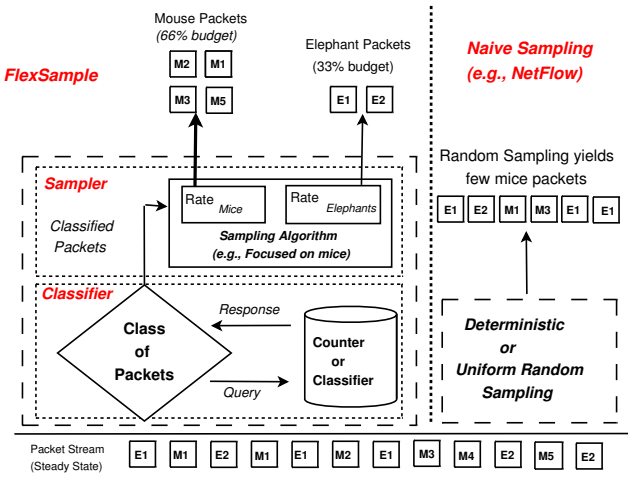


Figure 1: FlexSample vs. conventional sampling. E_i and M_i represent elephant and mouse packets, respectively.

As packets arrive, a lightweight counter classifies the packet according to an estimate of the size of the flow to which the packet belongs. The router then samples the packet with a probability that depends on the estimated class of the corresponding flow (*e.g.*, whether the packet belongs to a mouse flow or an elephant flow). FlexSample allows an operator to adjust the relative proportions of the sampling budget. To achieve these desired proportions, FlexSample computes and adapts the corresponding sampling probabilities for each class *online*.

1.2 Challenges

Allocating different sampling "budgets" to different classes presents several practical challenges both for counting and for sampling. The main goal for the **sampling** component is achieving the correct fraction of sampled packets for each class of traffic. Achieving this goal is difficult because neither the size of the flow to which each arriving packet belongs nor the traffic flow size distribution are known *a priori*. More subtly, achieving target proportions for each class requires working backwards to derive the appropriate sampling rates for each packet. To solve this problem, FlexSample computes rough initial sampling rates based on the target proportions and iteratively refines them; we find that this iterative refinement quickly converges to the correct sampling rates.

The **counting** component aims to accurately estimate the class of flow to which a particular packet belongs (*e.g.*, whether a flow is a mouse or an elephant), which is difficult to know *a priori*. To perform this classification in the face of dynamic traffic conditions (*i.e.*, as flows arrive and become inactive), FlexSample uses an array of n statistics counters, rather than a single counter. At any given time, $n-1$ counters are "warmed" and one counter is used for classification; the algorithm then cycles through the counters periodically, resetting each counter after it is used for classification. The number of counters in the array and the duration that each counter is used before rotation can be optimized according to flow arrival and departure characteristics.

1.3 Contributions

This paper makes the following contributions.

1. We explain the value of detecting small-volume traffic flows for two important classes of network management tasks—network security and application identification and classification—and quantify the extent to which existing sampling techniques fail to capture these flows.
2. We present a new framework for traffic sampling that offers significantly more flexibility in how sampling resources can be allocated across a traffic flow distribution. FlexSample is tunable with a few simple parameters: conventional random sampling corresponds to a specific setting of this parameter, but an operator can adjust this parameter to focus on a particular range of flow sizes (*e.g.*, capturing more unique "mice" vs. capturing more traffic from "elephants").
3. We provide a theoretical and empirical analysis of this framework, which demonstrates that significantly more flows can be captured for a relatively moderate decrease in the relative accuracy of estimates of the flow volumes. In particular, certain parameter settings of our algorithm allow for an accurate reconstruction of the network communication graph by capturing more low-volume flows than is possible with any existing technique.

FlexSample's general approach presents many possibilities for future work in high-speed traffic monitoring. Indeed, the main contribution of this paper is *not* to improve counting or sampling techniques, but rather to demonstrate how counter architectures and sampling algorithms can be used *together* to provide a more flexible traffic sampling framework. Our experiments incorporate a simple counter (a Counting Bloom Filter) and packet sampling scheme (random sampling), but our framework is general: It could incorporate *any* counter architecture and *any* sampling approach; for example, it could leverage the large body of previous work in algorithms for statistics counters [21, 25] and traffic sampling [10, 11].

Paper outline. Section 2 presents motivating scenarios for monitoring application structure on high-speed links. Section 3 presents an overview of the large body of related work in statistics counters and traffic sampling algorithms, much of which we believe the general FlexSample framework can leverage. Section 4 presents the FlexSample algorithm. Section 5 presents an analytical evaluation of FlexSample in terms of the unique flows it captures versus the relative error incurred in flow size estimation. Section 6 discusses implementation details and practical feasibility. Section 7 presents our evaluation of FlexSample on traffic traces from several operational networks. Section 8 describes how FlexSample can be extended both to more

than two traffic classes and for application-specific monitoring. Section 9 concludes.

2. Why Monitor Communication Structure?

Traffic monitoring has traditionally focused on the ability to monitor and track high-volume traffic flows; historically, this capability has been useful for many network operations tasks such as accounting [6, 8], monitoring for denial-of-service attacks [1, 2], etc. However, we believe that traffic monitoring applications require an increasing amount of flexibility, and the ability to track communication *structure*, as well as volume, is becoming increasingly important. In this section, we present two motivating scenarios for monitoring communication structure.

2.1 Tracking Threats

Communication networks have recently begun to see new types of unwanted traffic and other unconventional threats. Botnets are perhaps one of the most significant developing threats. Botnets (networks of compromised hosts under the control of a single entity) have been used to perpetrate denial of service (DoS) attacks, for spamming, and for other nefarious activities such as click fraud. Botnets are often controlled by a single "command and control" host, which compromised "bots", or machines, contact for further instructions. The control channel over which bots exchange messages with the command-and-control has historically been Internet Relay Chat (IRC), but variants may use other ports. The ability to observe more of a botnet's communication structure could reveal the identities both of the command-and-control and of the bots themselves. In Section 7, we demonstrate the extent to which naïve sampling strategies fail to capture such command-and-control structure and how FlexSample can more effectively recover this structure.

2.2 Traffic Classification

Monitoring communication structure may allow operators to classify application traffic based solely on these patterns, rather than having to rely on other potentially less indicative features (*e.g.*, the port number of the traffic flow). In their work on BLINC [17], Karagiannis *et al.* highlighted the importance and potential of such an approach: Fast, port-independent traffic classification can assist network operators with capacity planning and network design. It can also help operators monitor traffic trends for various applications in operational networks and distinguish various types of application traffic (*e.g.*, Web, peer-to-peer, streaming, attack) without looking at port numbers. This classification is most effective when the traffic being observed comprises a large number of source-destination pairs (*i.e.*, on a backbone link or links, rather than at the edge of the network), but traffic sampling can make this classification difficult.

2.3 Goal: Recovering Communication Structure from Sampled Traffic

Despite its potential, monitoring communication structure requires an *expansive network perspective*: The ability to monitor traffic at a location that is traversed by a large number of source-destination pairs (*i.e.*, a large backbone network). Unfortunately, these locations typically see immense volumes of traffic, thus preventing monitoring schemes from examining every packet. Because traffic flow sizes have a non-uniform distribution, uniform sampling will not recover traffic structure from Internet traffic. Our goal is to design a framework that intelligently reallocates a fixed sampling rate across flows to more effectively recover this structure.

3. Related Work

FlexSample draws on a large body of previous work in statistics counters and traffic sampling. Rather than contributing a new statistics counter or traffic sampling algorithm, FlexSample explores how lightweight statistics counters can better inform traffic sampling, given a fixed sampling constraint, to recover traffic *structure*. Though, in this paper, we implement FlexSample with a simple counter and naïve uniform sampling, the framework could potentially incorporate any of the statistics counters or sampling algorithms discussed below.

Traffic monitoring on high-speed links is typically performed with flow monitoring such as Cisco's NetFlow [23], Juniper traffic sampling [16], or InMon sFlow [15]. Because these techniques incur both high processing and collection overhead, routers must typically employ *packet sampling* on high-speed links. Packet sampling maintains statistics based on a sample of all packets that traverse the link; exported summaries about flows reflect the statistics of the sampled traffic. Traffic sampling inspects every nth packet—either deterministically or at random—using a configurable sampling technique, and continuously records statistics associated with the sampled packet's header in a local router cache until either a configured timeout value is reached or the cache is full, at which point the cache is flushed to a collector.

Stratified sampling divides traffic into equal-length strata and selects packets randomly from within the strata at a particular sampling rate [32]; this approach resembles FlexSample's division of sampling by epoch.

3.1 Size-Based Sampling

Size-dependent sampling has been proposed in two related contexts before: *flow sampling* and *packet sampling*. Size-dependent flow sampling deals with storage constraints on routers in cases where only a certain fraction of flow records can be retained; in this context, Duffield *et al.* proposed to sample and retain flow records with probability related to the original flow [7, 8]. Flow sampling—an offline decision as to whether an existing flow record should be stored—is fundamentally different from the online decision that FlexSample must make about whether to sample a packet.

The most closely related work to FlexSample is that of Kumar *et al.*, who propose using sketches to perform size-dependent packet sampling [20]. This approach, *sketch-guided sampling*, samples packets with a probability distri-

bution that depends on a the size of the flow to which the packet belongs. This work develops a general theory for how such sketches might be constructed and serves as the basis for the high-level FlexSample design. However, this previous work does not focus on capturing mouse flows, which is our primary goal in this work. Sketch-guided sampling focuses on a different class of problems (*e.g.*, tracking elephants, accounting, etc.), does not provide tunable parameters for capturing traffic flows in discrete size ranges or traffic classes, and does not consider traffic dynamics (*i.e.*, the fact that traffic distributions can change over time).

3.2 Inference and Tracking

Inference of traffic statistics. Previous work has devised methods to recover traffic statistics from sampled flow records with much success. Claffy *et al.* studied various sampling techniques at both packet-based or time-based granularities [5]. Others have attempted to improve sampling accuracy for estimating "heavy hitters", flow size distributions, traffic matrices, or packet flow arrivals for accounting, traffic engineering, or provisioning [4, 6, 9, 11, 13, 18, 19, 31]. Many of these techniques adapt the sampling rate to changes in flow characteristics, or attempt a different sampling strategy altogether; these techniques are chiefly concerned with drawing inferences about flow sizes or flow size distributions from sampled traffic statistics. In contrast, FlexSample uses a statistics counter to *control* the sampling process itself to solve a different class of problems (*i.e.*, recovery of traffic structure).

Application tracking and anomaly detection. Sampled traffic statistics have also been used to help operators detect malicious traffic [1, 2, 29], and many previous studies have demonstrated the utility of using sampled flow statistics for detecting high-volume attacks and malicious traffic [1, 9, 14]. However, more recent work has demonstrated that conventional sampling techniques can obscure statistics needed to detect traffic anomalies [3] or execute certain anomaly detection algorithms [22]. Previous traffic classification studies have used network communication structure to identify attack traffic [17, 29]. FlexSample can further assist operators with these tasks by allowing them to focus on specific flow size ranges (*e.g.*, mouse flows).

4. FlexSample

This section presents the FlexSample algorithm. We begin with an overview of FlexSample and a discussion of the challenges associated with the basic scheme (Section 4.1). We then present a detailed analysis of each of these problems and our approach to solving them (Section 4.2). Section 4.3 presents the algorithm itself, and Section 4.4 describes its properties. We defer discussions on the implementation of FlexSample, feasibility of using FlexSample on commodity routers, and enhancements to the basic FlexSample approach, to Sections 6.1, 6.2, and 8 respectively.

4.1 Algorithm Overview

At a high-level, FlexSample proceeds in two stages.

1. **Size estimation (Counting).** For each packet, obtain an approximate size for the flow it belongs to, using a lightweight counter that only considers fixed fields in the network and transport headers (*e.g.*, the packet's {source IP, source port, destination IP, destination port, protocol} 5-tuple).
2. **Sampling.** Sample the packet at a probability based on its flow size estimate. FlexSample allows an operator to specify the proportions of the sampling budget to allocate to each flow-size range (also referred to as a "flow-size bin", "bin", or a "sampling class"). FlexSample then computes and adapts the corresponding sampling probabilities for each class *online* to achieve these proportions.

Simple as this approach appears on the surface, a closer inspection brings the following challenging questions to bear:

C_1 *How can the counter be architected to be fast, memory-efficient, and accurate*?

C_2 *How can the counter combat pollution of flow size estimates due to the presence of long-inactive flows*?

C_3 *How can the traffic percentages for each flow-size bin (e.g., the percentage of all traffic for flows that contain between 5 and 10 packets) be predicted* a priori*, and how can these be used to compute packet sampling probabilities*?

The next section addresses each of these challenges.

4.2 Challenges for Flexible Sampling

C_1**: A fast, accurate counter architecture.** The FlexSample counter is an array of Counting Bloom Filters (CBFs). Each CBF is keyed by flow-unique identifiers in a packet and maintains approximate statistics on the size of each flow. CBFs [12] are widely used as approximate-matching memory-efficient data structures that support both insertion and removal. Well-understood derivations exist for calculating the number of entries required given the memory available and a target error rate. Section 6.1 explains the CBF parameters we chose; in that section, we also show that even if this counter cannot afford to sample every packet, it can, with high probability, still correctly classify flows.

C_2**: Aging inactive flows.** To ensure that the counter values accurately represent flow sizes, we build an *array* of CBFs ("CBFArray"), rather than using a single CBF. As Figure 2 shows, the CBFs in the CBFArray are used in a staggered fashion. The flow-key for a new packet is inserted into *every* CBF; an entry is created if one did not previously exist; otherwise, the counts are incremented. Only the *oldest* CBF is used to classify flows. After one *epoch* (which can be defined in terms of time or packets seen), we clear the CBF currently being used and begin using the next CBF in the array. This process allows FlexSample to efficiently and gracefully age inactive flows without requiring a timer.

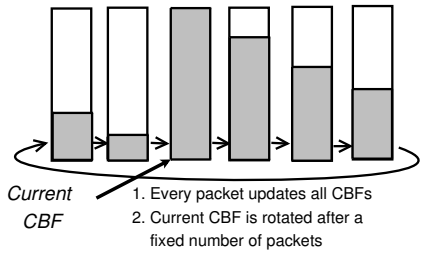


Figure 2: Staggered Usage of CBFs in the CBFArray. Shaded regions indicates the extent to which a CBF has been "warmed" with insertions/updations. The CBF with the longest history is used for lookups at any time.

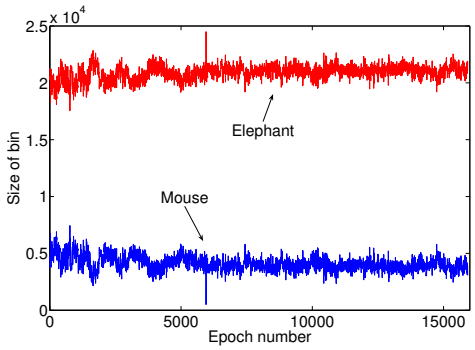


Figure 3: The total number of packets sampled for each class of flow is reasonably stable across epochs, which implies that future sampling probabilities can be based on historical observations. It also implies that our analysis for approximating the total number of unique flows captured can assume constant values of γ_m and γ_e for the entire trace duration. (For this experiment, $T = 50$.)

C_3**: Estimating future traffic percentages, and calculating per-bin sampling probabilities.** Since the CBFArray already keeps counts for flows, estimating the current fraction of traffic for each bin is straightforward. With a few additional counters, one can maintain the fractions in real-time without requiring extra lookups on the CBFArray. However, FlexSample requires an estimate for the *future* fraction of traffic from a flow-size bin, f_i; to compute this estimate, FlexSample uses an Exponential Weighted Moving Average (EWMA) on current and past estimates of traffic fractions, giving higher weight to the latest estimates. Our analysis of the variation in traffic fractions, presented in Figure 3, affirms our choice of using historical information to predict future fractions of traffic.

4.3 Algorithm Details

Figure 4 shows the FlexSample algorithm in detail; Table 1 defines notation. First, the operator provides: The fraction of sampling budget to allocate for each class, α_m and α_e, and a flow size threshold, T, which indicates the demarcation between mouse and elephant flows. For instance, setting $\alpha_m = 0.5$ and $T = 10$ indicates that the operator wants 50% of sampled packets to be allocated to mouse flows, and every flow smaller than 10 packets to be considered a mouse. An appropriate setting for T depends on the actual traffic flow size distribution; knowledge of this distribution makes it easier to set a value of T that suits an operator's objectives, but we find in Section 7 that FlexSample performs well for a wide variety of settings. Given these parameters, FlexSample must compute initial instantaneous sampling rates for each flow-size bin, γ_m and γ_e; these values are initially set as shown in Figure 4.

Variable	*Definition*
T	The flow-size threshold separating an elephant from a mouse.
s	The sampling rate for naïve random sampling.
α_m	The fraction of mice the operator wishes to obtain in the final sampled output.
α_e	The fraction of elephants the operator wishes to obtain in the final sampled output.
f_m	The fraction of traffic estimated to be mice.
f_e	The fraction of traffic estimated to be elephants.
γ_m	The *instantaneous* sampling probability for the mouse class.
γ_e	The *instantaneous* sampling probability for the elephant class.

Table 1: Summary of notation.

Next, as packets arrive, FlexSample uses flow size estimates provided by a counting bloom filter array (CBFArray) to continually reallocate its *fixed* sampling budget (*i.e.*, sample, on average, no more than one out of every N packets) to regions of the flow-size distribution that a network operator is most interested in.

The CBFArray supports two functions: (1) ESTIMATEFLOWSIZE, which accepts a key that uniquely identifies a flow and returns the CBFArray's appraisal of the number of packets it has witnessed from that flow; and (2) ESTIMATEFRACTION, which, given a sampling class, returns an estimate of the fraction of prior traffic that falls within that class. Ideally, we require an estimate of *future* traffic; we use an Exponential Weighted Moving Average (EWMA), with higher weight (75%) to the latest estimate of traffic fractions. Both functions are constant-time operations that require at most one lookup or computation. For now, we assume that the CBFArray returns a reasonably accurate estimate (*i.e.*, has infinite storage, and automatically ages out stale flows); Section 6.1 discusses the practical considerations in implementing the CBFArray.

Finally, FlexSample updates its instantaneous sampling rates for each class of flow, based on its estimates of how many packets of each type were captured in the previous epoch. Once α_m, α_e, T, and the original sampling rate s are configured, FlexSample calculates the instantaneous sampling probabilities (γ_m and γ_e for mice and elephants, respectively) for each sampling class according to the equations:

$$\gamma_m = \frac{\alpha_m \times s}{f_m}; \qquad \gamma_e = \frac{\alpha_e \times s}{f_e}$$

where f_m is the instantaneous estimate of the fraction of mouse traffic, and f_e is the instantaneous estimate of the fraction of elephant traffic. FlexSample computes f_m and f_e using the counters in the CBFArray and updates them whenever a mouse or elephant packet is looked up. Appendix A describes the algorithm that FlexSample uses to calculate f_m and f_e "on the fly".

The appropriate settings for T depend on both the flow

```
class CountingBloomFilterArray:
    var Array of Counting Bloom Filters
    function EstimateFlowSize(packet p):
        return the count of packets for the flow p belongs to
    endfunction
    function EstimateFraction(sampling "class" S):
        return the sampling "class"
            (i.e., whether Elephant or Mouse) of p
    endfunction
    function Update(packet p):
        update internal counters to account for p
    endfunction
end class
```

```
procedure FlexSample ():
Input:
    Packet Stream; P
    Threshold; T
    Original Sampling Rate; s
    Sought fractions of Mice and Elephants
        in sampled traffic; α_m, α_e
Output:
    Sampled Elephant Packets; E
    Sampled Mouse Packets; M
Algorithm:
CBF := new CountingBloomFilterArray
// Initially assume equal traffic fractions for both bins
let γ_m := 2α_m s; γ_e := 2α_e s
for each packet p from stream P:
    // Query the CBF to estimate this flow's class
    let c := CBF.EstimateFlowSize(p)
    // Sample the packet according to its class
    if c < T; then
        sample p with probability γ_m into M
        // Re-estimate the fraction of mice packets in P
        let f_m := CBF.EstimateFraction('mice');
    else
        sample p with probability γ_e into E
        // Re-estimate the fraction of elephant packets in P
        let f_e := CBF.EstimateFraction('elephant')
    end if
    // Recalculate sampling probabilities for each class
    γ_m := (s × α_m)/f_m ; γ_e := (s × α_e)/f_e
    // Update CBF with current packet's information
    CBF.Update(p)
end for
```

Figure 4: Definitions of the CountingBloomFilterArray **structure and the** FlexSample **procedure.**

size distribution and the packet arrival rate (Refer to the Section 5 for a discussion about tuning α_m). Without any knowledge of the flow size distribution or arrival process, the best setting of T is the sample mean for a single epoch: the setting of T for the entire distribution, divided by the number of epochs in the trace for which the distribution was measured. However, if flow arrivals are bursty (as is often the case with wide-area traffic [24], as well as the traces we use in our evaluation), then flows tend to be contained within one or two epochs, and the best setting of T is the desired setting of T for the distribution over the entire trace.

4.4 Properties

FlexSample has the following properties, which we will explain further in Sections 5 and 7.

Property 1 *FlexSample neither underflows nor overflows the sampling budget.*

If $\alpha_m + \alpha_e = 1$, the expected number of packets sampled at steady state is equal to the number of packets sampled by uniform random sampling at rate s.

Property 2 *For appropriate settings of α and T, FlexSample captures more unique flows without substantially increasing the relative error for flow size estimation.*

FlexSample offers provable bounds on the error introduced due to reduction of sampling budget for large flows. We discuss these bounds in Section 5 and show that they hold in our empirical evaluation in Section 7. Our results in Section 7 also show that FlexSample can also be used to recover specific traffic structures, such as botnet "command and control" traffic.

Property 3 *FlexSample automatically adjusts instantaneous sampling probabilities to satisfy* post facto *per-bin packet ratios.*

The per-bin sampling probabilities γ_m and γ_e react to changes in traffic distribution (reflected in values of f_m and f_e), and readjust to meet each bin's overall sampling budget.

Property 4 *FlexSample can be extended to support multiple sampling classes.*

As we show in Section 8, FlexSample can be extended to any number of sampling classes with different target requirements (*i.e.*, α values) for each class, simultaneously meeting the individual per-bin budgets as well as the overall budget. As in the 2-bin case, FlexSample adjusts the instantaneous sampling probabilities (*i.e.*, γ values) for each bin on the fly.

5. Inferring Properties of Original Traffic

Operators use statistics from sampled flows to estimate characteristics of the original traffic. Extrapolating from samples incurs error. In our work, we consider two types of error incurred by FlexSample: 1) the number of unique flows, and 2) the size of each flow. (Conventional sampling schemes are typically concerned with only the latter, but because we are also interested in recovering network structure.) In this section, we derive analytical expressions for each type of error incurred by FlexSample. These expressions can help an operator understand both how complete the graph representing traffic structure is (*i.e.*, how many flows are missing from the trace) and the accuracy of the flow size estimates.

5.1 Determining Fraction of Unique Flows

Recovering communication structure requires capturing as many unique flows as possible. We define a flow as the communication between a pair of hosts represented by the 5-tuple {source IP address, source port number, destination IP address, destination port number and protocol}. In this section, we show that FlexSample can capture more unique

flows than naïve sampling (this analysis is also supported by our experiments in Section 7). We also present basic guidelines for tuning FlexSample's parameters to sample a certain fraction of unique flows.

Naïve sampling samples each packet with constant probability. In this case, determining the expected number of unique flows captured is straightforward. Let:

$$\begin{aligned} s &= \text{Sampling probability} \\ F(r) &= \text{Number of packets in flow } r \\ R &= \text{Set of all flows} \\ R_{mouse} &= \text{Set of all mouse flows} \\ R_{elephant} &= \text{Set of all elephant flows} \\ \mathcal{U} &= \text{Number of unique flows in the actual trace} \end{aligned}$$

The probability that at least one packet from flow r is sampled is: $P(r) = [1 - (1 - s)^{F(r)}]$. Given a list of flows and the flow size distribution, it is easy to determine the expected number of unique flows sampled. Let $\widehat{\mathcal{U}}$ represent the number of flows actually sampled by a particular strategy. For naïve sampling, this value is:

$$\widehat{\mathcal{U}}_{\text{naïve}} = \sum_{r \in R} 1 \text{ flow} \times P(r) = \sum_{r \in R} [1 - (1 - s)^{F(r)}]$$

Lower sampling probabilities imply that a smaller number of unique flows are captured. The nonlinearity in the relation between $\widehat{\mathcal{U}}$ and $F(r)$ indicates that a mouse flow with few packets is highly unlikely to be sampled, a characteristic that is unfortunate for capturing network structure.

In contrast, FlexSample classifies each packet based on the estimated bin k of its corresponding flow and applies simple random sampling at a corresponding probability p_k. In the two-bin case, each packet belonging to a mouse flow is sampled at a uniform probability of γ_m, and an elephant packet is sampled at γ_e. As mentioned in Section 4, these sampling probabilities are recomputed *online* in every epoch to react to changes in the traffic distribution and can potentially vary.

Let us assume that mouse flows are sampled with constant probability throughout the entire trace. Figure 3 shows that the total number of packets ("bin size") for mice and elephants is fairly constant across epochs. This characteristic implies that γ_e and γ_m will not vary too greatly between epochs, so this assumption is reasonable for the purposes of our analysis.

This assumption allows us to show that FlexSample captures more unique flows than naïve sampling. The expected number of unique flows observed with FlexSample, $\widehat{\mathcal{U}}_{\text{FlexSample}}$, is:

$$= \sum_{R_{mouse}} [1 - (1 - \gamma_m)^{F(r)}] + \sum_{R_{elephant}} [1 - (1 - \gamma_e)^{F(r)}]$$

$$= \sum_{R_{mouse}} [1 - (1 - \frac{\alpha_m \times s}{f_m})^{F(r)}] + \sum_{R_{elephant}} [1 - (1 - \frac{\alpha_e \times s}{f_e})^{F(r)}]$$

Computing the number of unique flows captured using naïve sampling yields:

$$\widehat{\mathcal{U}}_{\text{Naïve}} = \sum_{R_{mouse}} [1 - (1 - s)^{F(r)}] + \sum_{R_{elephant}} [1 - (1 - s)^{F(r)}]$$

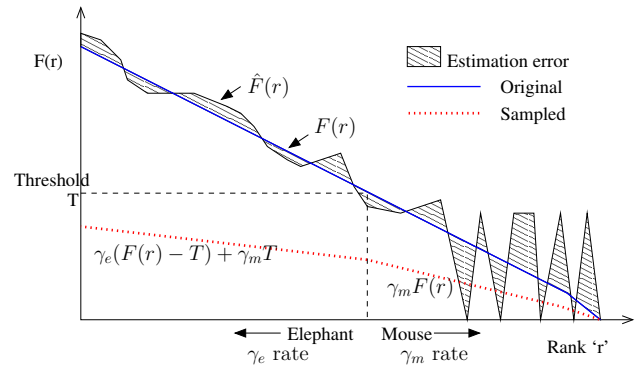


Figure 5: Illustration of two-bin FlexSample and the flow size estimation during a particular time window. $\widehat{F}(r)$ represents the estimated flow size distribution, while F(r) represents the original flow. Note that the two distributions are in log-scale.

Comparing these two expressions shows that the first term in $\widehat{\mathcal{U}}_{\text{FlexSample}}$ is greater than the first term in $\widehat{\mathcal{U}}_{\text{Naïve}}$ when value of α_m be greater than f_m. Furthermore, the second term in the expression typically tends to 1, regardless of the value of the elephant sampling rate. This is because the $F(r)$ for all elephant flows is at least over the threshold value T, thereby causing the value $(1 - \gamma_e)^{F(r)}$ to be very small and ensuring that each elephant flow is sampled at least once[1]. Decreasing the threshold T can potentially lead to a decrease in the number of elephant flows sampled. However, this decrease in the number of unique elephant flows captured is more than compensated by an increase in the number of unique mouse flows captured. Thus, as long as the value of α_m is chosen appropriately, FlexSample captures more number of unique flows than naïve sampling. Accordingly, we recommend the following rule of thumb: *Select an α_m that is greater than the fraction of mouse packets in the overall trace*.

5.2 Recovering Flow Size Estimates

FlexSample generates sampled flow records, along with the instantaneous sampling probability each flow was subject to. Based on this value, we can compute an estimate of the actual flow size and then compute the *relative estimation error*. We define: (1) *estimation error*: the difference between the actual number of packets present in the trace and the estimated flow size, and (2) *relative estimation error*: the estimation error normalized over the actual flow size.

Estimating flow size is challenging because the sampling parameters vary across different epochs. To determine the total flow size estimate, we must compute the flow size estimate per epoch and sum these over the entire trace. After classifying which bin k a particular flow belongs to, FlexSample samples a packet that flow at some probability p_k. If FlexSample samples $n_i(r)$ packets of a particular flow r of size $F_i(r)$ during epoch i, then the flow size estimate $\widehat{F}_i(r)$ is $\left\lfloor \frac{n_i(r)}{p_k} \right\rfloor$.

[1]For example, for a γ_e value of 0.01, any flow with more than 458 packets will be sampled with probability 0.99.

Figure 5 illustrates the flow size estimation process associated with FlexSample. $\widehat{F}(r)$ represents the estimated flow size distribution and $F(r)$ represents the original flow. Note that there are cases when a flow is never sampled; these cases yield an estimated flow size of zero. We consider such cases to be an error both in the number of unique flows sampled and in the flow size estimate.

The sampling probability $\widehat{F}_i(r)$ is not a single constant factor: rather, it depends on the type of the flow (mouse or elephant in the case of a 2-bin sampling scheme), which in turn depends on the information in the classifier and the threshold value T. Thus, for a 2-bin sampling process, we can express the flow size estimate during the epoch i as:

$$\widehat{F}_i(r) = \begin{cases} \left\lfloor \frac{n_i(r)}{\gamma_m} \right\rfloor, & \text{if mouse} \\ \\ \left\lfloor \frac{n_i(r)}{\gamma_e} \right\rfloor, & \text{if elephant} \end{cases}$$

If a flow becomes an elephant flow during the course of an epoch, FlexSample must handle this case carefully so that an operator can later determine which packets were sampled at rate γ_m and which were sampled at γ_e. To handle this transition, FlexSample logs the sampling rate of each bin during each epoch. An operator can later determine the number of packets in each bin and sum up the estimate for each bin. The expression for the flow estimate in the 2-bin case is:

$$\widehat{F}_i(r) = \left\lfloor \frac{n^{mouse}(r)}{\gamma_m} + \frac{n^{elephant}(r)}{\gamma_e} \right\rfloor$$

Logging this information enables an operator to compute the flow size estimate without much concern for the type of the flow and other epoch-specific information. The flow size estimate can now be easily computed as:

$$\widehat{F}_i(r) = \left\lfloor \sum_{j=0}^{n_i(r)} \frac{1}{\text{Sampling rate of pkt } j \text{ in epoch } i} \right\rfloor$$

In a similar manner, we can compute the estimation error per epoch and then add these errors to determine the overall estimation error for the flow. Estimation error $E_i(r)$ can be expressed as $F_i(r) - \widehat{F}_i(r)$, which depends on the number of packets sampled during each epoch. Let $X(r) = \{X_1(r), X_2(r), ...X_i(r), ...\}$ be the random process representing the number of packets of flow r sampled during each epoch i. For each flow r, the probability that we sample $n_i(r)$ packets is given by a binomial distribution. This probability is given as follows:

$$P(X_i(r) = n_i(r)) = \binom{F_i(r)}{n_i(r)} p_k^{n_i(r)} (1 - p_k)^{F_i(r) - n_i(r)}$$

The expected value for the flow size estimation error $\bar{E}_i(r)$ for the particular flow r during epoch i is:

$$\begin{aligned} \bar{E}_i(r) &= \sum_{n_i(r)=0}^{F_i(r)} P(X_i(r) = n_i(r)) \times (F_i(r) - \widehat{F}_i(r)) \\ &= F_i(r) - \sum_{n_i(r)=0}^{F_i(r)} P(X_i(r) = n_i(r)) \times \widehat{F}_i(r) \\ &\approx \text{Rounding error} \end{aligned}$$

Thus, the estimation error follows a binomial distribution with mean in the range $[0, 1)$, because of the inherent rounding error. This also indicates that FlexSample will be able to offer acceptable estimates of the traffic by keeping track of the instantaneous sampling rates. We recommend setting the value of α_m in the range of $(0, 1)$, because setting α_m to 0 or 1 can cause one of the traffic classes to be completely ignored in the sampling process (causing higher estimation error).

Over a certain time interval Δ when a flow r is active, the overall estimation error $E(r)$ is $E(r) = \sum_{i=0}^{\Delta} E_i(r)$ and the overall relative estimation error in flow size is $RE(r) = \frac{E(r)}{F(r)}$. Figure 5 shows certain mouse flows for which the estimation error, and therefore the relative estimation error, is a large negative value, which can happen when the flow size estimate computed from the sampled packets is significantly bigger than the original flow size. Such overestimation can sometimes be problematic for traffic engineering or provisioning.

Much of the discussion in this section has focused on improving detection and estimation of individual flows. When an operator wants to estimate the overall traffic volume (for traffic engineering and resource provisioning), a more relevant metric is the *relative estimation error in volume*, or the relative estimation error in the total trace, which can be expressed as:

$$\text{Rel. est. error in volume} = \frac{\sum_{r \in R} F(r) - \widehat{F}(r)}{\sum_{r \in R} F(r)}$$

6. Implementation

In this section, we describe the implementation of FlexSample (Section 6.1), and discuss the feasibility of implementing FlexSample in router hardware (Section 6.2).

6.1 Counter Implementation

We implemented FlexSample in approximately 1,000 lines of C++ code. The chief design choices concern the implementation of the counter. We use a staggered array of Counting Bloom Filters (*CBFArray*) for counting the number of packets in each flow for a particular time interval.

CBFArray Implementation. The CBFArray trades off space (due to redundancy in entries: there exist multiple copies of the same key among the elements in the CBFArray because we perform inserts to all CBFs in the array) for efficiency. Staggering multiple arrays and periodically rotating

them expires inactive flows without requiring timers. We believe that this approach is more efficient than a timer-based approach. First, timer interrupts over 100KHz can potentially result in up to 45% overhead for the processor merely in responding to interrupts [28]. Second, one timer would be required for *each* entry in the counter; therefore, timer updates must access *every* element in the counter.

In contrast, the staggered configuration of CBFs with periodic rotation ensures that no flow that has been inactive for a rotation through the array will exist in the counter. Careful choice of the number of CBFs and the CBF rotation condition (*i.e.*, a time interval or witnessing a certain number of packets) can provide reasonably accurate emulation of timer expiry without its concomitant overhead. Rotation based on number of packets seen is easier to implement and can never overshoot the target number of CBF entries (because the CBF will always be rotated after a fixed number of packets) even at very high packet rates (*e.g.*, a DoS attack); therefore, we use this scheme in our implementation.

Using the target CBF error rate and total SRAM available on the system, we can compute the overall number of entries the CBFArray can accommodate using standard Bloom Filter calculations [12]. The actual number of CBFs used depends on the traffic distribution: A trace dominated by short or medium-sized flows may use a larger number of CBFs (with more frequent rotation, to quickly expire flows that become inactive), while a trace containing mostly long-lived large-volume flows should use fewer CBFs to save memory.

Our implementation (which we use for our evaluation in Sections 7 and Section 8) used a CBFArray with 4 CBFs, each of which accommodates 100,000 entries with 0.01 error rate. Each entry in the CBF is a 1-byte counter, and the largest memory footprint for this structure (measured by the maximum resident set size of the program) was 5,156 KB.

Hashing. FlexSample computes hashes over its flow-specific fields, which are in turn used for the CBF lookup and updation. We call the ratio of the number of packets hashed to the total number of packets seen as the *hashing rate* k. For the classification to yield meaningful information, $k \gg s$ (the sampling rate) [2]; $k = 1$ in our experiments. Fast hashes over well-defined bytes of a packet can be performed at very high speeds [30] (making hashing every packet practically feasible), but even if the counter cannot examine every packet, we argue that the classifier should still perform well.

Claim 1 *With high probability, a counter with a hashing rate of $k < 1$ will not misclassify elephants as mice (or vice versa).*

Proof Sketch. Let flows with size greater than or equal to T_e be called elephants and those less than T_m, mice. $T_e \geq T_m$. Let the observed arrival rate for elephants be μ, and that for mice be $\mu - 2\delta$ for some $\delta > 0$.

[2]Note that if $k < 1$, the packets that get looked up must be sampled at an overall rate of s/k to maintain the total sampled traffic size.

If $k < 1$, we can use Chernoff bounds to show that, if packets arriving at a rate of at least $\mu - \delta$ are classified as elephants and those arriving at a rate of at most $\mu - \delta$ are classified as mice, then the probability of misclassifying elephants as mice (or vice versa) is *less* than $e^{-2\mu\delta^2}$. ■

6.2 Feasibility of a Hardware Implementation

Compared to conventional sampling techniques, FlexSample requires: (1) a significant amount of fast memory (SRAM); and (2) the ability to compute packet hashes at near-line speeds. We offer why both requirements are reasonable, at least for today's high-end routers.

SRAM limits. SRAM on routers can be on-chip or off-chip. According to Varghese ([28, page 441]), on-chip SRAM has latencies below 5ns, and is limited to about 64 megabits. Off-chip SRAM latencies around 10ms, but often have higher capacities. In all our experiments, the maximum memory used by the CBFArray structure was less than 6 MB, *i.e.*, less than 48 megabits, which is well within the SRAM limits in 2004 when Varghese reported the trend.

In 2001, Sanchez *et al.* introduced a technique of storing large lookup data structures using limited fast memory (SRAM) for frequently accessed data and expansive slow memory (DRAM) for storing data that is not currently used [26]. Data is transferred to and from the DRAM in bulk, but not often enough to cause a performance bottleneck. A slight modification to the CBFArray structure allows it to use this paradigm, guaranteeing constant SRAM usage for any number of CBFs. Recall from Section 4.2 that only the CBF with the longest history from the CBFArray is used for lookups at any time, while the rest are merely "warmed". Thus, FlexSample could store *only* the CBF that is currently being looked up in SRAM. Because the *same* insertions are applied to all CBFs in the array in any one epoch, we merely need some additional memory (less than the size of 1 CBF) in SRAM to record the insertions that happened in the current epoch; these can be applied *all at once* to the next CBF in sequence when it is brought in to SRAM at the end of the current epoch.

Hashing at line speeds. Bloom filters use multiple independent hash functions to reduce chances of conflict. The exact number of functions required to query or insert a key into a Bloom filter is dependent on the size of the filter and the target error rate; in our case, for a size of 100,000 entries and target error rate of 0.01, FlexSample requires 7 hash functions.

Although our evaluation uses a software implementation of the SHA-1 hash that is difficult to compute quickly in hardware [27], in theory, *any* universal hash function would suffice for the CBF. Recent research has shown that hardware implementations of the linear congruential hash (LCH) universal hash function have a throughput of over 10 Gbps [30].

7. Evaluation

In this section, we evaluate FlexSample on multiple traffic

traces to demonstrate its ability to recover communication graph structure without incurring prohibitive flow size estimation error.

7.1 Data

Packet traces from campus and ISP networks. Our primary dataset is a traffic capture from the ingress link of a large campus network. The trace lasts for around 2.5 hours on April 4, 2006 and comprises approximately 398 million packets. We process the trace to extract the flow-key (*i.e.*, a concatenation of the flow-specific 5-tuple) before sending the output to FlexSample[3]. We repeated our experiments on a traffic trace from a small ISP's uplink on January 28, 2004; this trace contains approximately 8.8 million packets over about 1 hour.

Packet traces with "ground truth" mouse flows. As described in Section 2, botnet "command-and-control" (C&C) communication is one example of a set of mouse flows for which a network operator might like to observe general structure. To examine FlexSample's effectiveness for extracting traffic structure for this specific type of traffic, we generated a traffic trace with known botnet C&C communication and measured the extent to which FlexSample could extract this communication. Because we do not have botnet C&C traces for the timeframe of our primary packet capture, we devise the following strategy to intersperse botnet C&C traffic within legitimate traffic. We extract the portion of botnet C&C traffic data between 7:30 a.m. and 10 a.m. EST on November 22, 2005 (approximately 1.2 million packets)—the same local time and day of the week as our primary traffic trace—and merge it with our primary trace by adding an appropriate constant offset to all timestamps in the botnet trace. We also marked the injected packets to allow us to later identify which packets belonged to the botnet trace.

7.2 Results

In this section, we show that FlexSample captures more unique flows than naïve sampling for only a modest increase in flow size estimation error. We observe that, except in extreme cases where the sampling budget is allocated either all to elephants or all to mice (*i.e.*, $\alpha_m = 0$ or 1), estimation error is roughly equivalent to that of naïve sampling.

Result 7.1 (Unique Flows Captured) *When optimized to capture mouse flows, FlexSample captures up to twice as many unique flows than naïve sampling does.*

FlexSample aims to detect as many unique flows as possible to expose communication *structure*. Figure 6 shows the number of unique flows obtained with FlexSample for various settings of T and the target sampling fraction for mice, α_m (refer to Table 1 for notation). Also shown is the baseline for naïve random sampling (which does not depend on α_m). We set the sampling rate s for all experiments to 0.01.

[3]The last 9 bits of the source and destination IP addresses in the trace are zeroed to preserve user privacy, but we believe that considering the source and destination port information minimizes the chances that our experiments would consider multiple flows as a single flow.

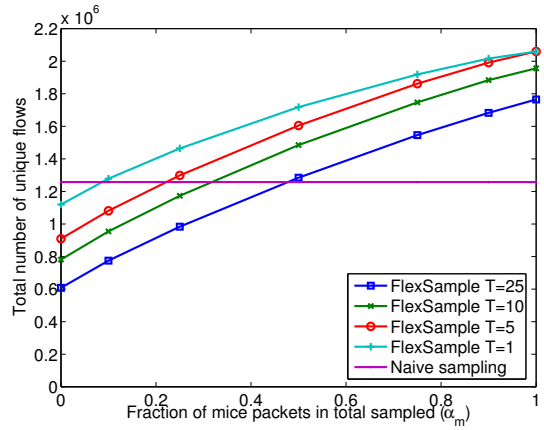


Figure 6: The number of unique flows captured by FlexSample for different settings of α_m and threshold T, along with the baseline performance of naïve sampling. The base sampling rate s was 1/100 in this experiment.

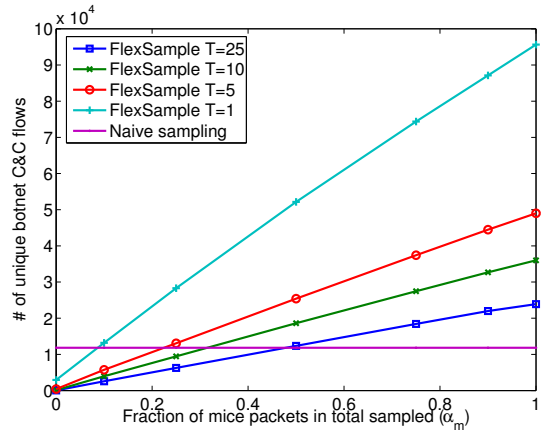


Figure 7: Number of unique Botnet C&C flows discovered with various values of the threshold T, in comparison to those discovered with naïve sampling. The base sampling rate s was 1/100 in this experiment.

With $\alpha_m = 1$ and $T = 1$, FlexSample captures over 2 million unique flows, about 64% more than with naïve sampling. Even increasing T to 25 (at $\alpha_m = 1$) captures nearly 50% more flows than the naïve sampling. Finally, even if α_m is low, a low value of T will *still* capture more unique conversations than the baseline. Therefore, network operators willing to set apart even 20% of their sampling budget for mice could capture almost 16% more unique flows than the baseline, while incurring minimal additional estimation error for larger flows.

We repeated our experiments on the second trace and observed similar results: the number of flows captured with FlexSample with $\alpha_m = 0.9$ and threshold $T = 1$ was 2.89 times that captured with naïve sampling; the mean relative estimation error for elephants was approximately equal to that of naïve sampling.

Result 7.2 (Botnet Flows Captured) *On a trace with*

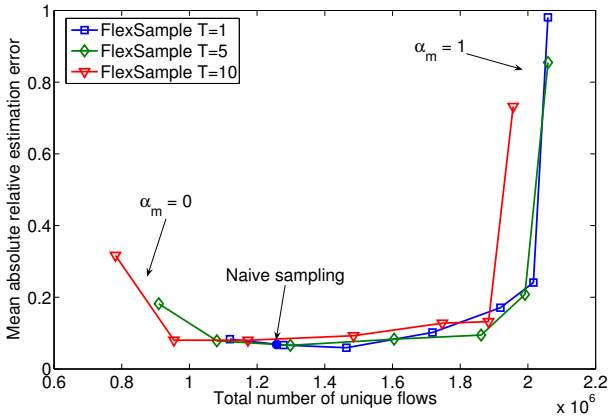


Figure 10: The mean of relative estimation error incurred over the whole trace by flows in the size range of [10000, 10100] packets, for different settings of α_m and threshold T. Note that the number of unique flows captured increases as α_m increases.

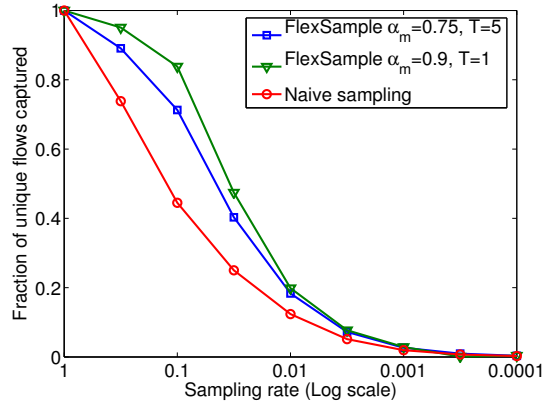


Figure 11: Effect of sampling rate on the number of unique flows captured with naïve sampling and FlexSample.

known botnet traffic, FlexSample exposed more than eight times as many botnet flows as naïve sampling.

To investigate whether FlexSample captures *known* mouse flows for a specific type of application, we repeated these experiments with botnet traffic "mixed in" (Section 7.1). The botnet trace alone contains about 1.2 million packets belonging to 361,129 unique flows. Figure 7 plots the results of the experiment for this trace. Even low values of α_m expose more unique flows than naïve random sampling; a setting of $T = 1$ and $\alpha_m = 1$ exposes more than eight times as many flows (95,656) as naïve sampling—almost one-fourth of all unique botnet flows.

Result 7.3 (Estimation Error) *FlexSample captures significantly more unique flows than naïve sampling without affecting flow size estimation error for large flows.*

Identifying mouse flows is an added feature of a sampling technique, so any scheme that captures more unique flows than naïve sampling must minimize flow size estimation error. Figure 10 plots the mean value of the relative estimation error incurred over the *whole* trace by a certain set of flows that have a flow size between [10000, 10100] packets, for different settings of α_m and T. The estimation error of flows in that size range does not degrade significantly for most values of $\alpha_m \neq 0$ or 1; this characteristic implies that α_m can be increased within limits to capture significantly more unique flows, without incurring any additional relative error. For example, a setting of $T = 5$ yields an additional 45 million flows—50% more than naïve sampling—with negligible increase in estimation error. Similar trends occur for other elephant flow size ranges.

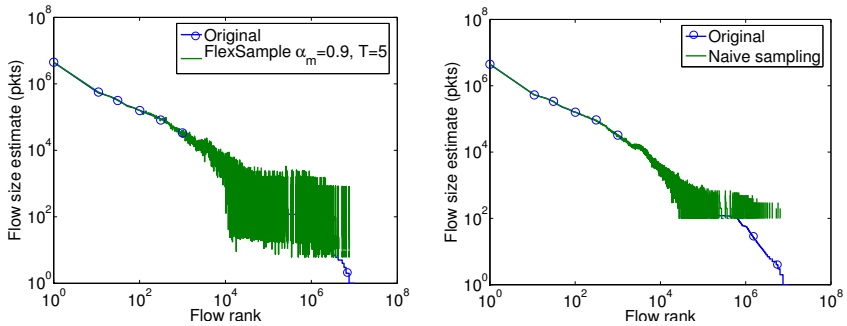
Figure 8 explains why high α_m settings result in large estimation error.[4] When a higher fraction of the sampling budget is allocated to mouse flows for small T, the instantaneous sampling probability for the mouse bin increases. The low threshold causes even very small flows to be sampled, potentially even multiple times. In such cases, the calculated estimate could be greater than the actual flow size even by a few orders of magnitude. The spikes in Figure 8 (a) for small and medium size flows are due to overestimation; this figure also shows that the additional error relative to naïve sampling is negligible for all flow sizes greater than 10^3.

The relative estimation error incurred in the total volume with FlexSample was no worse than naïve sampling. Additionally, FlexSample yields acceptable error values for all parameter settings, except for boundary cases of $\alpha_m = 0$ or 1. However, the large variation in the estimate of individual flows (as observed from Figure 8) causes the estimate of the total volume to be unreliable. Thus, network operators often use SNMP counters for these kind of aggregated traffic information. An easier way to determine the estimate of the total volume with FlexSample would be to determine the total number of samples generated and dividing it by the base sampling rate, s. FlexSample also satisfies the overall sampling constraint: the number of samples that FlexSample captures is approximately the total number of packets times the sampling rate s.

Figure 9 shows the cumulative distribution of the absolute relative estimation error incurred per flow for FlexSample versus naïve sampling. The relative estimation error for the two sampling techniques are quite similar for almost 90% of the flows. Allocating 90% of the sampling budget to capture mouse flows causes the estimate to be skewed in the remaining 10% of the flows. Furthermore, FlexSample incurs large relative estimation error for mouse flows in comparison to naïve sampling. This error arises because FlexSample captures a higher proportion of mouse flows and incurs a large over-estimation error, while naïve sampling fails to discover many of the mouse flows.

Result 7.4 (Effect of Sampling Rate) *The difference in the number of flows captured with FlexSample is greatest at a sampling rate of 0.1.*

[4] As explained in Section 5, the boundary values for α_m are not suitable choices in the light of estimation error.



(a) FlexSample (b) Naïve sampling

Figure 8: Comparison of the distribution of flow size estimates for FlexSample and naïve sampling. "Flow rank" refers to the rank in the reverse-sorted list of flow sizes.

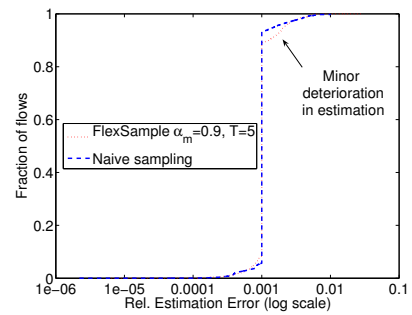


Figure 9: Cumulative distribution of the relative estimation error incurred by individual flows with FlexSample and naïve sampling.

For the majority of our experiments, we chose a sampling rate of 0.01; many large networks (*e.g.*, Abilene, the US research and educational network), use this rate. Figure 11 compares the effect of sampling rate on FlexSample's ability to capture unique flows. FlexSample captures more unique flows than naïve sampling, irrespective of the overall sampling rate s. Further, the difference in number of unique flows captured by FlexSample and by naïve sampling is greater between high (*e.g.*, about 0.7) to medium (*e.g.*, 0.01) sampling rates. As the sampling rate approaches 1, mouse packets have a high probability of being sampled even with naïve sampling; thus, the advantage of the FlexSample technique becomes less apparent. At low sampling rates (0.001 and below), a large value for α_m does not yield significantly more unique flows than naïve sampling because the instantaneous sampling probability γ_m is proportional to s (which is very small), so the other parameters (f_m and α_m) have no effect (see Appendix A). Since the most commonly used sampling rates range between 0.1 to 0.001, FlexSample, with appropriate settings, can capture significantly more flows than naïve sampling.

8. Extending FlexSample: Multiple Classes

We have extended FlexSample to support more than two classes with almost no additional overhead. The only required modification is that the instantaneous traffic fractions, f_i, and instantaneous sampling probabilities, γ_i, must now be maintained for more than two classes. The CBFArray is independent of the number of sampling bins. In this section, we explain how "multi-bin" FlexSample can be used both to focus sampling on sub-populations within the traffic distribution (Section 8.1) and to capture more unique flows *for a specific application* by allocating sampling budgets to the most common flow-size ranges for that application (Section 8.2).

Algorithm	# unique flows	Mean rel. est. error
3-bin FlexSample ($\alpha_1 = 0.75, T_1 = 1$ $\alpha_2 = 0.1, T_2 = 50$)	1.87×10^6	0.1717
2-bin FlexSample ($\alpha_m = 0.9, T = 1$)	2.016×10^6	0.241
Naïve sampling	1.257×10^6	0.06819

Table 2: Unique flows and mean relative estimation error of flows with size in the range [10000, 10100], obtained for 3-bin FlexSample.

8.1 Focusing on Sub-Populations

Operators can configure FlexSample with multiple thresholds to concentrate their sampling budget *only* on very small or very large flows, ignoring the medium-sized flows. More generally, an operator can set low sampling allocations to classes which are deemed to be uninteresting, focusing instead on sub-populations of interest (*e.g.*, recovering communication structure from mouse sub-populations) to achieve better tradeoffs between recovering communication structure and minimizing flow size estimation error.

For certain parameter settings, multi-bin FlexSample captures more unique flows than naïve sampling *and* maintains a lower overall error rate than the two-bin configuration: a desirable middleground. We configured a 3-bin FlexSample experiment, with 75% of the sampling budget (α_1) allocated to flows that have a size of 1 packet in each epoch and 15% of the budget allocated (α_3) to flows with size over 50 packets in each epoch. We then compared the number of unique flows captured, as well as the relative estimation error for flows in the size range $[10000, 10100]$ against the 2-bin configuration from Section 7. Table 2 shows the results of this analysis. This 3-bin configuration provides better flow size estimates for elephant flows than the 2-bin configuration (a relative error of 0.1717, compared with 0.241 in the 2-bin case), without losing the improvement in number of unique flows captured.

8.2 Application-Specific Communication

An operator may wish to recover the communication structure for a particular flow or application, which would necessarily require maximizing the number of unique flows of a particular protocol or application. Multi-bin FlexSample can help an operator recover structure for a specific type of application traffic, presuming that the operator knows the flow size range or ranges where the traffic of interest commonly falls. For example, Figure 12 shows the flow-size distributions for traffic to and from three common ports—53, 80, and 22 (corresponding mostly to the applications

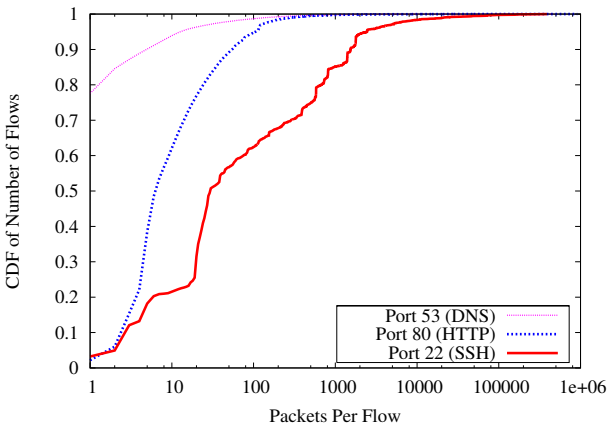


Figure 12: Flow size distributions of three common ports corresponding to DNS (port 53), HTTP (port 80), and SSH (port 22).

Algorithm	All flows	HTTP flows
3-bin FlexSample ($\alpha_1 = 0.2, T_1 = 1$ $\alpha_2 = 0.7, T_2 = 10$)	1.864×10^6	1.014×10^6
2-bin FlexSample ($\alpha_m = 0.9, T = 1$)	2.016×10^6	0.922×10^6
Naïve sampling	1.257×10^6	0.622×10^6

Table 3: Unique HTTP flows inferred with 3-bin FlexSample.

DNS, HTTP, and SSH respectively). An operator could set thresholds T_i and sampling proportions α_i to focus precisely on ranges where traffic for a particular application was most concentrated.

In our traffic trace, about 80% of HTTP flows have flow sizes between 1 and 10. Based on this observation, we configured a 3-bin FlexSample setup where flows of size 1 were allocated a budget of 20% ($T_1 = 1, \alpha_1 = 0.2$) and flows with size between (0, 10] packets in each epoch received a modest budget of 70% ($T_2 = 10, \alpha_2 = 0.7$).

Table 3 compares the number of unique flows captured by 3-bin FlexSample against 2-bin FlexSample and naïve sampling. Focusing on the flow size range where HTTP traffic is highly concentrated helps capture more unique HTTP flows, thus providing a more effective recovery of communication structure for HTTP traffic. Specifically, we captured 9.95% more unique HTTP flows. Refining multi-bin FlexSample to more effectively focus on application-specific communication structure is an area for future work.

9. Conclusion

This paper has presented FlexSample, a flexible framework for traffic monitoring that allows a network operator to skew sampling rates towards a particular range of the traffic distribution. FlexSample is based on a simple premise: An approximate, lightweight counter provides hints about interesting packets to a resource-constrained sampling process that can capture more detailed statistics, yet cannot afford to look at every packet. In this paper, we use FlexSample to study an important, yet understudied problem: recovering traffic *structure* (*i.e.*, "Who's talking to whom?"). Previous work has demonstrated the utility of recovering traffic structure [17], but this task requires capturing significantly more unique flows than is possible with naïve sampling techniques (*e.g.*, [16, 23]), which capture a disproportionate number of packets from flows with large traffic volumes. Most previous work has focused on modifications to traffic sampling algorithms that accurately track flow sizes for "heavy hitter" flows (*e.g.*, for accounting or traffic engineering purposes). In contrast, appropriate settings of FlexSample can capture significantly more unique "mouse" flows while incurring negligible error for estimates of traffic flow sizes.

By (1) sampling flows of different sizes at different rates and (2) using a lightweight counter to provide hints to the sampling process about the size of the flow to which the packet belongs, FlexSample can capture at least 50% more unique flows than naïve random sampling at the same sampling rate for a negligible increase in relative estimation error on traffic flow sizes. FlexSample takes as input the fraction of packets that should be captured for each class (*i.e.*, size range) of traffic flows and threshold values to demarcate each size range; it then computes the instantaneous sampling probabilities for each traffic class on the fly to achieve the appropriate *post facto* ratios of sampled traffic. Our analytical and empirical evaluation demonstrates that FlexSample is capable of recovering significantly more unique flows than naïve traffic sampling; further, our experiments indicate that FlexSample might be used to recover specific traffic structures of interest, such as botnet "command and control" traffic patterns. Finally, we have demonstrated with empirical evaluation that FlexSample can be extended to support differentiated sampling rates over more than two traffic flow size ranges.

The design of FlexSample can offer more general lessons for traffic monitoring on high-speed links. Specifically, FlexSample has demonstrated that, in practice, using a small, lightweight classifier can significantly provide hints to a monitoring process that can examine traffic in closer detail, yet has tighter resource constraints. We believe that this general approach can be used to perform fine-grained monitoring of traffic subsets of high-speed links.

Acknowledgements

We thank Russ Clark and Dave Andersen for the traces used in our analysis, and Anukool Lakhina for many inspiring discussions that laid the groundwork for this paper. We also thank Muhammad Mukarram Bin Tariq, Fabian Monrose, Jim Xu, Abhishek Kumar, and Sushant Rewaskar for comments that helped improve the paper.

REFERENCES

[1] Arbor Networks. `http://www.arbornetworks.com`.

[2] P. Barford, J. Kline, D. Plonka, and A. Ron. A Signal Analysis of Network Traffic Anomalies. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[3] D. Brauckhoff, B. Tellenbach, A. Wagner, A. Lakhina, and M. May. Impact of Traffic Sampling on Anomaly Detection Metrics. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Rio de Janeiro, Brazil, Oct. 2006.

[4] B.-Y. Choi and S. Bhattacharyya. On the Accuracy and Overhead of Cisco Sampled NetFlow. In *Proceedings of ACM SIGMETRICS Workshop on Large Scale Network Inference (LSNI)*, June 2005.

[5] K. C. Claffy, G. C. Polyzos, and H.-W. Braun. Application of sampling methodologies to network traffic characterization. In *Proc. ACM SIGCOMM*, pages 194–203, San Francisco, CA, Sept. 1993.

[6] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proc. ACM SIGCOMM*, pages 325–336, Karlsruhe, Germany, Aug. 2003.

[7] N. Duffield, C. Lund, and M. Thorup. Predicting resource usage and estimation accuracy in an IP flow measurement collection infrastructure. In *Proc. ACM SIGCOMM Internet Measurement Conference*, pages 179–191, Miami, FL, Oct. 2003.

[8] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley. Inferring Link Loss Using Striped Unicast Probes. In *Proc. IEEE INFOCOM*, Anchorage, AK, Apr. 2001.

[9] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.

[10] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[11] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, Aug. 2003.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, pages 254–265, Vancouver, Canada, Sept. 1998.

[13] N. Hohn and D. Veitch. Inverting sampled traffic. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Miami, FL, Oct. 2003.

[14] Y. Huang and J. Pullen. Countering Denial of Service Attacks using Congestion Triggered Packet Sampling and Filtering. In *Proceedings of International Conference on Computer Communications and Networks*, pages 490–494, 2001.

[15] InMon sFlow. `http://www.inmon.com/technology`.

[16] Juniper traffic sampling and forwarding overview. `http://www.juniper.net/techpubs/software/junos/junos71/swconfig71-policy/html/sampling-overview.html`.

[17] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *Proc. ACM SIGCOMM*, pages 229–240, Philadelphia, PA, Aug. 2005.

[18] R. Kompella and C. Estan. The Power of Slicing in Internet Flow Measurement. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[19] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. ACM SIGMETRICS*, pages 177–188, New York, NY, June 2004.

[20] A. Kumar and J. Xu. Sketch Guided Sampling – Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *Proc. IEEE INFOCOM*, Barcelona, Spain, Mar. 2006.

[21] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement. In *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004.

[22] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is Sampled Data Sufficient for Anomaly Detection? In *Proc. ACM SIGCOMM Internet Measurement Conference*, Rio de Janeiro, Brazil, Oct. 2006.

[23] Cisco NetFlow. `http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html`.

[24] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[25] S. Ramabhadran and G. Varghese. Efficient Implementation of a Statistics Counter Arhictecture. In *Proc. ACM SIGMETRICS*, San Diego, CA, June 2003.

[26] L. A. Sanchez, W. C. Milliken, A. C. Snoeren, F. Tchakountio, C. E. Jones, S. T. Kent, C. Partridge, and W. T. Strayer. Hardware Support for a Hash-Based IP Traceback. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX)*, 2001.

[27] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid To Network Processing. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[28] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers Inc., 2004.

[29] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applcations. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[30] B. Yang, R. Karri, and D. A. McGrew. Divide and Concatenate: An Architectural Level Optimization Technique for Universal Hash Functions. In *Proceedings of the Design Automation Conference*, San Diego, CA, 2004.

[31] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An Information-Theoretic Approach to Traffic Matrix Estimation. In *Proc. ACM SIGCOMM*, pages 301–312, Karlsruhe, Germany, Aug. 2003.

[32] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall. *Sampling and Filtering Techniques for IP Packet Selection, Internet-Draft, draft-ietf-psamp-sample-tech-07.txt, Work in Progress*, 2005.

Appendix

A. Deriving Instantaneous Sampling Rates

FlexSample classifies packets into different sampling bins and then applies instantaneous sampling rates to each class of flow (*i.e.*, in the two-bin case, "mouse" and "elephant"), according to two parameters that the operator sets: the *post facto* proportion of mouse packets, α_m, and a threshold value T which separates mice from elephants. Given these values, FlexSample must calculate the *instantaneous* rate at which to sample a mouse or elephant flow (γ_m and γ_e, respectively). The instantaneous sampling rates are subject to two constraints:

- **Budget constraint:** The total number of sampled packets must be equal to that sampled by naïve sampling at a rate of s. If FlexSample samples $n_i(r)$ packets from flow r of size $F_i(r)$ during epoch i, then the following constraint must hold:

$$s \times \sum_R F(r) = \sum_{i=0}^{M} \sum_R n_i(r) = \sum_{i=0}^{M} \left(\sum_{R_{mouse}} n_i(r) + \sum_{R_{elephant}} n_i(r) \right)$$

  where M is the total number of epochs in the trace.

- **Distribution constraint:** The instantaneous sampling rate must be tuned such that each bin contributes the right proportion of packets to the overall *post facto* fraction of packets in each bin. If tot_m and tot_e represent the number of packets belonging to all mouse flows and elephant flows, respectively, then we must ensure that:

$$\text{Total sampled} = tot_m \cdot \gamma_m + tot_e \cdot \gamma_e$$

The definition of α_m gives the following condition:

$$\alpha_m = \frac{\sum_{i=0}^{M} \sum_{R_{mouse}} n_i(r)}{\text{Total sampled}}$$
$$= \frac{tot_m \cdot \gamma_m}{tot_m \cdot \gamma_m + tot_e \cdot \gamma_e}$$

The above equations yield:

$$\alpha_m = \frac{tot_m \cdot \gamma_m}{s \cdot (\text{Total original})}$$
$$\Longrightarrow \gamma_m = \frac{\alpha_m \cdot s \cdot (\text{Total original})}{tot_m}$$
$$= \frac{\alpha_m \cdot s}{f_m}$$

Similarly, $$\gamma_e = \frac{\alpha_e \cdot s}{f_e}$$

where f_m and f_e represent, respectively, the fraction of mouse packets and elephant packets in the trace.

Because it is impractical to predict the future values of f_m and f_e, FlexSample relies on historical information to estimate these values using EWMA.