Fast Monitoring of Traffic Subpopulations

Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, and Vijay Vazirani
School of Computer Science, Georgia Tech
266 Ferst Drive, Atlanta, GA, USA
{avr,srini,feamster,vazirani}@cc.gatech.edu

ABSTRACT

Network accounting, forensics, security, and performance monitoring applications often need to examine detailed traces from subsets of flows ("subpopulations"), where the application requires flexibility in specifying the subpopulation (*e.g.*, to detect a portscan, the application must observe many packets between a source and a destination with one packet to each port). Unfortunately, the dynamism and volume of network traffic on many high-speed links requires traffic sampling, which adversely affects subpopulation monitoring: because many subpopulations of interest to operators are low-volume flows, conventional sampling schemes (*e.g.*, uniform random sampling) can miss much of the subpopulation's traffic. Today's routers and network devices provide scant support for monitoring *specific* traffic subpopulations.

This paper presents the design, implementation, and evaluation of *FlexSample*, a traffic monitoring framework that dynamically extracts traffic from subpopulations that operators define using conditions on packet header fields. *FlexSample* uses a fast, flexible counter array to provide rough estimates of packets' membership in respective subpopulations. Based on these coarse estimates, *FlexSample* then makes per-packet sampling decisions to sample proportionately from each subpopulation (as specified by a network operator), subject to an overall sampling constraint. We apply *FlexSample* to extract subpopulations such as port scans and traffic to high-degree nodes and find that it can capture significantly more packets from these subpopulations than conventional approaches.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks]: Network Monitoring C.4 [Computer-Communication Networks]: Measurement Techniques

General Terms: Algorithms, Design, Measurement, Security

Keywords: traffic subpopulations, traffic statistics, sampling, counters, FlexSample

1. INTRODUCTION

Routers and other devices that monitor traffic on high-speed networks cannot collect and record accurate statistics based on every packet in a traffic stream. Updates to statistics typically require access to a large—and relatively slow—memory (such as DRAM). They thus rely on packet *sampling*: selecting packets uniformly at random from the packet stream. Uniform random sampling works well for the traditional goals of billing and traffic engineering because these metrics require accurate estimates of only the "heavy-hitter" flows, which are typically well-represented in sampled traffic. Although many schemes have improved upon uniform random sampling [14, 15, 43], heavy-hitter identification remains their primary focus.

Recently, however, operators have started to monitor traffic for a broader range of applications: identifying P2P "supernodes", servers with many clients, infected computers (bots) engaged in activities such as spam, denial-of-service, or portscanning. Much of this traffic consists of small-volumed flows that have few packets per flow ("mouse" flows). Because techniques such as uniform random sampling select more packets from heavy-hitter flows, they will likely miss the presence of small-volume flows. Thus, these conventional or "naïve" sampling techniques are less appropriate for these new monitoring applications.

There is need for a technique that can sample packets not just from heavy-hitter flows (such as uniform random sampling), but also from other traffic *subpopulations*: subsets of flows that have some common property or behavior. To capture more packets from certain subpopulations, the packet selection algorithm needs to efficiently determine whether a packet in the stream belongs to that subpopulation and bias its sampling rate to ultimately capture more or less traffic from the respective subpopulation. To accomplish this goal, we presents *FlexSample*, a framework and technique to bias packet selection towards certain subpopulations of traffic, subject to an overall sampling constraint. *FlexSample* provides expressiveness and flexibility beyond naïve packet sampling techniques used by existing systems such as Cisco's Sampled NetFlow [31]. The key idea behind *FlexSample* is that high-speed network devices can maintain approximate statistics using fast, space-efficient counters to determine the subpopulation to which each packet belongs; these counters can then be used to bias packet selection towards packets that belong to desired subpopulations.

FlexSample allows an operator to specify the characteristics of traffic subpopulations (*e.g.*, packets from flows that have less than 10 packets, packets from a source IP address that has sent over 100 packets, etc.), as well as a *sampling budget*—the fraction of the expected number of sampled packets obtained using the original sampling rate—that is set apart for packets selected from each subpopulation. *FlexSample* selects packets that belong to different subpopulations at different *instantaneous* sampling probabilities such that: (1) the overall expected number of packets selected is equal to the expected number of packets selected by uniform random sampling; (2) the fraction of sampled packets belonging to each subpopulation are according to sampling budgets specified by the operator. *FlexSample* meets both of these constraints using counters maintained in fast memory (SRAM) that are updated at every packet arrival.

Operators can specify traffic subpopulations *FlexSample* using an expressive, simple configuration language using conditions that are based on conjunctions of counter values. For example, consider a source IP address $srcip$ performing a portscan attack on destination $dstip$. An operator who wants to capture more packets that correspond to this behavior can specify a *FlexSample* condition for an artifact of scanning behavior observable in the packet stream. For a portscan, *FlexSample* maintains two counters: the first counter simply counts the tuple ($srcip, dstip$), *i.e.*, the number of packets sent from $srcip$ to $dstip$, while the second counter counts the tuple ($srcip, dstip, dstport$) (*i.e.*, the number of packets sent from $srcip$ to $dstip$ for the destination port on the *current* packet). The condition for a packet to be identified as part of a portscan could be, for example, "the first counter has a value greater than 30 *and* the second counter has a value less than 2" (if the attacker sends only one probe packet per destination port). This condition will be matched for all but the first 30 packets; the subsequent will be sampled at a higher probability than they would using naïve sampling.

1.1 Design and Scope

Figure 1 presents the high-level design of *FlexSample* using the portscan example above. *FlexSample*'s operation comprises two stages: *counting* and *sampling*. In the counting stage, *FlexSample* looks up counters for certain sets of fields of the packet header (called "*tuples*"). In the sampling stage, *FlexSample* uses the counts and sampling budgets specified by the operator to calculate an *instantaneous* probability at which the current packet is to be sampled. The current packet is then sampled at that probability.

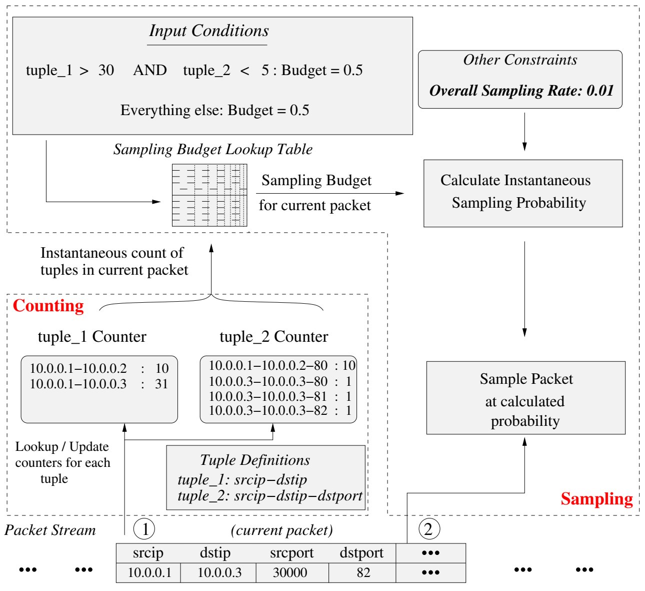
Scope. *FlexSample* captures more packets from certain subpopulations by reducing the effective sampling rate for other traffic (including packets from heavy-hitter flows). Although this reduction typically increases the error in estimation flow sizes for heavy-hitter flows (a common application for traffic engineering and accounting), we show in Section 7 that the additional error incurred over naïve sampling is typically small. Moreover, we expect that the network operator will typically want to monitor small-volume flows for short periods of time (*e.g.*, during a spam or a DoS attack); in this case, the extra error incurred on estimates of heavy-hitter flows over the long term will be negligible.

FlexSample can preferentially select traffic subpopulations only if the subpopulation's characteristics can be expressed using tuple-counts. Some subpopulation characteristics (*e.g.*, time-shifted behavior, such as periodic probes) cannot be captured using tuple-counts alone, and *FlexSample* will typically not be useful in this scenario. However, as we show in Section 6, conditions on tuple-counts are powerful enough for a variety of monitoring applications: *FlexSample* was able to capture *many times more packets* from small-volume subpopulations such as portscans and botnet "command-and-control" traffic.

1.2 Contributions

This paper presents the following contributions.

- A new framework that uses a small amount of fast memory to preferentially select packets from small-volume traffic *subpopulations* based on conditions that can be specified as conjunctions of counter values, such that overall sampling constraints are always maintained. The framework represents a decoupling of packet *selection* into a general two-stage approach. The first stage, *counting*, provides a coarse-grained estimate for the desirability of any packet using a cheap, fast computation (possibly using fast counters). The second stage, *sampling*, combines the first-stage estimates with physical constraints (*e.g.*, storage space, line-speed, etc.) and decides whether to select the packet.
- A simple, expressive configuration language that allows operators to specify conditions to bias packet selection towards certain subpopulations, and an array of counters that allows on-the-fly updates of the conditions themselves.
- An implementation and evaluation of *FlexSample* on real traces and several motivating applications.



Input Conditions
tuple_1 > 30 AND tuple_2 < 5 : Budget = 0.5
Everything else: Budget = 0.5
Other Constraints
Overall Sampling Rate: 0.01
Sampling Budget Lookup Table
Sampling Budget for current packet
Calculate Instantaneous Sampling Probability
Instantaneous count of tuples in current packet
Counting
tuple_1 Counter
tuple_2 Counter
10.0.0.1−10.0.0.2 : 10
10.0.0.1−10.0.0.3 : 31
10.0.0.1−10.0.0.2−80 : 10
10.0.0.3−10.0.0.3−80 : 1
10.0.0.3−10.0.0.3−81 : 1
10.0.0.3−10.0.0.3−82 : 1
Sample Packet at calculated probability
Lookup / Update counters for each tuple
Tuple Definitions
tuple_1: srcip−dstip
tuple_2: srcip−dstip−dstport
Sampling
Packet Stream
1
(current packet)
2
srcip dstip srcport dstport
10.0.0.1 10.0.0.3 30000 82

Figure 1: High-level design of *FlexSample*, showing the two stages of the packet selection process. The counting stage involves looking up (and simultaneously updating) counters according to pre-specified tuple definitions. The specifications also include the input conditions (here, the operator has allocated 50% of the sampling budget to packets which match the portscan rule). The sampling stage uses the tuple-counts to calculate an instantaneous probability of sampling each packet.

Our evaluation also demonstrates that *FlexSample*'s flexibility comes at a reasonable cost: it can capture higher fractions of certain "interesting" small-volume flows (such as bot traffic, super-sources, or super-destinations), while simultaneously satisfying more conventional traffic monitoring objectives, such as estimation of heavy-hitter flow sizes.

The rest of the paper is organized as follows. Section 2 presents an overview of the *FlexSample* algorithm. We describe *FlexSample*'s counting scheme in Section 3 and sampling algorithm in Section 4. Section 5 discusses implementation details and practical feasibility. Section 6 evaluates *FlexSample* on traffic traces from the Georgia Tech network. We discuss *FlexSample*'s impact on flow size estimation error in Section 7. Section 8 describes how *FlexSample* can be extended to support different types of applications, or network operator needs. Section 9 presents an overview of related work, and Section 10 concludes.

```
# base sampling rate
sampling_rate = 0.01
# number of tuples
tuples = 2
# number of conditions
conditions = 1
# tuple definitions
tuple_1 := srcip.dstip
tuple_2 := srcip.srcport.dstport
# condition :  sampling budget
tuple_1 in (30, ∞] AND
    tuple_2 in (0, 5]:  0.5
```

Figure 2: *FlexSample* input specification file for preferentially sampling portscan packets. Lines beginning with '#' are comments.

2. OVERVIEW

As Figure 1 shows, the *FlexSample* framework consists of: (1) a language the operator can use to specify desired subpopulation characteristics in terms of conditions on counts of tuples (Section 2.1); (2) a set of counters that maintain up-to-date counts of the tuples (Section 2.2); and (3) a sampling algorithm (and associated data structures) that use counter values to decide which packets to select from the input stream (Section 2.3).

Section 2.4 describes how traffic variability affects *FlexSample*.

2.1 Specification Language

FlexSample's specification language defines: (1) the number of tuples that must be counted, (2) the sets of header fields that constitute each tuple, (3) conditions on tuple-counts, and sampling budgets to be allocated to each.

Figure 2 shows a simplified input specification file for the portscan example of Figure 1. The specification instructs *FlexSample* to count two tuples: (1) $(srcip, dstip)$, which indicates the total number of packets sent from source $srcip$ to destination $dstip$, and (2) $(srcip, dstip, dstport)$, which indicates the number of packets from $srcip$ to the current packet's destination port ($dstport$). The final line of the specification provides the condition on tuple-counts for the desired subpopulation. Conditions are written as conjunctions of clauses, where each clause is of the form $tuple \in range$; in this case, the condition is: the count of packets from $srcip$ to $dstip$ is at least 30, *and* the count of packets to the current packet's $dstport$ is less than 5. The condition also has an associated sampling budget—0.5 in this case—which implies that *FlexSample* should tune packet sampling probabilities such that 50% of *sampled* packets matched this condition.

Challenge. Notice that the conditions only specify the sampling budgets for desired subpopulations; *FlexSample* must also compute sampling budgets for packets that do not match any of the specified conditions. Even for the single condition in Figure 2, there are four combinations of tuple ranges—there are two ranges for each tuple—of which only one combination has an explicit sampling budget specification. For combinations of tuple ranges without explicitly specified budgets, the budgets must be deduced. In addition, certain conditions may not contain range specifications for all tuples (*e.g.*, if the condition in Figure 2 had not specified a range for `tuple_2`, *i.e.*, "`tuple_1 in (30, ∞]:  0.5`"); in such cases, the subpopulation corresponding to the condition is unaffected by (*i.e.*, "*dont-care*") the values of unspecified tuples.

Solution. *FlexSample* needs to quickly lookup the sampling budget for *every* possible combination of tuple ranges at run-time. Thus, in a pre-processing step, *FlexSample* converts the conditions in the input specification into a data structure called the *sampling budget lookup table*. The sampling budget lookup table is an m-dimensional structure (called m-D ARRAY)—where m is the number of tuples—that contains a unique sampling budget for every possible combination of tuple *ranges*. Each dimension of the structure corresponds to one of the tuples being counted, and the number of rows in a dimension is equal to the number of mutually exclusive and exhaustive ranges for the tuple corresponding to the dimension. More formally, suppose $v_1, ..., v_m$ are the m tuples and $|v_k|$ represents the number of mutually exclusive and exhaustive ranges the tuple v_k can have (deducible from the input specification). Further, let $T_{(v_i,1)}, ..., T_{(v_i,|v_i|+1)}$ be the set of disjoint ranges for tuple v_i (where the first element, $T_{(v_i,1)}$, is 0 and the last element, $T_{(v_i,|v_i|+1)}$, is ∞). Then, the entry b of the sampling budget lookup table indexed by the m ranges $(T_{(v_1,i_1)}, T_{(v_1,i_1+1)}], ..., (T_{(v_m,i_m)}, T_{(v_m,i_m+1)}])$ (where $i_k \leq |v_k|; 0 < k \leq m$) would be the budget specified by the condition:

```
tuple_1 in (T_(v1,i1), T_(v1,i1+1)] AND
  tuple_2 in (T_(v2,i2), T_(v2,i2+1)] AND
   ...
    tuple_m in (T_(vm,im), T_(vm,im+1)] :   b
```

We call these entries—each of which corresponds to a unique conjunction of tuple ranges—*sampling classes*. Thus, the total number of sampling classes is equal to the product of the number of distinct ranges for each tuple. The sampling budget lookup table is a discrete density function in m dimensions, with each entry specifying the target fraction of packets for one set of tuple ranges and the total sum of all entries equalling 1.

		tuple_2	
		(0, 5]	(5, ∞]
tuple_1	(0, 30]	*0.167*	*0.167*
	(30, ∞]	**0.5**	*0.167*

Table 1: The sampling budget lookup table constructed using the specification from Figure 2. The bold entry is an explicitly specified budget while the italicized entries are deduced (using Scheme 1: dividing the leftover sampling budget equally among unspecified sampling classes).

Table 1 shows a 2-dimensional sampling budget lookup table constructed from Figure 2. The bold entry corresponds to the sampling class (*i.e.*, the conjunction of tuple ranges) with an explicitly specified sampling budget; the remaining unspecified sampling budgets are *deduced* using the constraint that the sum of all entries in the sampling budget lookup table must equal 1. There are two schemes for deduction: (1) divide the remaining unspecified sampling budget *equally* among the unspecified sampling classes; or (2) sample at the same probability from each unspecified sampling class such that the total number of sampled packets from these classes equals the remaining sampling budget (note that, in this case, the remaining sampling budgets will not be divided equally). Scheme 1 provides an equal number of sampled packets from each unspecified sampling class, while Scheme 2 provides a uniform random sample from packets that belong to any of the unspecified sampling classes.

2.2 Subpopulation Counters

FlexSample maintains one counter per tuple; these are incremented at each packet arrival[1]. The counters count "keys"

[1]As we sketch in Appendix A, as long as the counters are updated at a higher rate than the overall sampling rate, *FlexSample* will still be able to preferentially select packets from desired subpopulations (albeit with higher error).

comprising concatenation of fields of the tuple (*e.g.*, the tuple $(srcip, dstip)$ for a packet with $srcip$ 10.1.1.1 and $dstip$ 10.1.1.2 will be counted using the key "10.1.1.1-10.1.1.2").

Challenges. First, because routers have limited fast memory, *FlexSample* cannot precisely count each tuple and must use *approximate* counters; the challenge is to achieve an acceptable tradeoff between the accuracy of the counts and their memory consumption. Second, because the entities being counted—packet header fields—are representative of a dynamic traffic mix, the counter must somehow *age out* stale counter values.

Solutions. To meet these challenges, *FlexSample* uses a counter structure that consists of an array of Counting Bloom filters [16], called a *CBFArray*. Counting Bloom filters (CBFs) are space-efficient approximate counters that can be configured to meet an operator's accuracy or memory constraints. Our extension to CBFs—the CBFArray—uses multiple CBFs in a configuration that allows aging out stale counter values *without requiring explicit timers per counter entry*. Section 3 presents the counter design in detail.

2.3 Preferential Sampling

After the counts of each tuple are retrieved and counters updated, *FlexSample* makes a decision about whether to sample the current packet or not based on an *instantaneous sampling probability*. The probability calculation first determines the sampling budget allocated to the current packet by looking up the budget corresponding to the counts of input tuples obtained from the current packet.

Challenge. Calculating the instantaneous sampling probability requires meeting two constraints: (1) The fractions of packets sampled in each subpopulation must correspond to the operator's sampling budget specifications, which we call the *local* constraint, and (2) The effective sampling rate (*i.e.*, the fraction of all packets sampled) must be equal to the naïve sampling rate, or the *global* constraint.

Solution. The local constraint depends on the fraction of packets of each subpopulation in traffic *on the wire*, which may fluctuate from time to time. *FlexSample*'s challenge is to modulate instantaneous sampling probabilities such that the two constraints above are always satisfied. *FlexSample* solves the problem using a bookkeeping structure that keeps track of the *instantaneous* traffic mix, and an intuitive algorithm to calculate instantaneous sampling probabilities for each packet (Section 4).

2.4 Traffic Dependence

Varying fractions of subpopulations in traffic also presents challenges for *FlexSample*: (1) the sampling budgets an operator chooses for a subpopulation may not always match the percentage of sampled packets that belong to the desired subpopulation; (2) choosing sampling budgets to achieve a "good" sample of the desired subpopulation requires knowledge of the actual traffic percentages of these subpopulations. We show how the first problem can affect *FlexSample* in Section 6.1.1. Although the current design of *FlexSample* does not include solutions for either challenge, we discuss potential solutions for these issues in Sections 8.1(a) and 8.4, respectively.

3. COUNTER DESIGN

CBFs 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 [16].

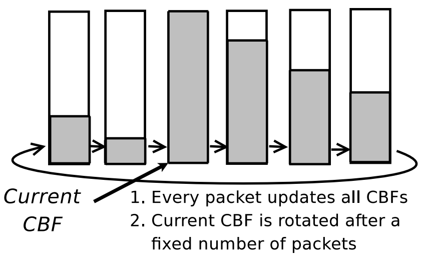

Current CBF
1. Every packet updates all CBFs
2. Current CBF is rotated after a fixed number of packets

Figure 3: Staggered use of CBFs in the CBFArray. Shaded regions indicates the extent to which a CBF has been "warmed" with insertions/updates. The CBF with the longest history is used for lookups at any time.

As Figure 3 shows, CBFs in the CBFArray are used in a staggered fashion. For insertion, the key is added into *every* CBF: an entry is created if one did not previously exist; otherwise, the count of the entry is incremented. Although all CBFs are updated, only the count from the *oldest* CBF is used for processing. Periodically (where periods can be defined in terms of time or of packets seen), *FlexSample* clears the CBF currently being used and begins using the next CBF in the array; we call these periods *epochs*. This "rotation" process allows *FlexSample* to efficiently expunge stale information due to inactive or dead entries without maintaining timers *per entry*. The epoch length, size of each CBF, resolution of each counter in a CBF entry, and the number of CBFs in a CBFArray, all depend on the characteristics of the tuples that are counted by the CBFArray; as a rule-of-thumb, ROTATE is called when the current CBF is near its capacity, or if the counts of entries in the CBF have grown stale. The CBFArray parameters may also be tuned using the *FlexSample* input specification file. Section 5.1 explains the default CBFArray parameters we chose.

The CBFArray supports three simple functions: (1) LOOKUP, which accepts a key—a concatenation of a set of fields in the packet header—and returns the CBFArray's appraisal of how many times that key has been seen in the *oldest* CBF; (2) INCREMENT, which accepts a key and increments the counters for the key in *all* CBFs in the array; and (3) ROTATE, which is called at the end of each epoch and causes the CBFArray to flush the current CBF and update its internal pointers to start reading from the next-oldest CBF.

In our current implementation, INCREMENT is typically called as part of LOOKUP because lookups on the CBFArray occur only when a packet with the specified key is seen in the packet stream, which implies that the key will be inserted subsequently. All operations are constant time and computationally cheap. Section 5.1 discusses the actual CBFArray implementation.

4. SAMPLING ALGORITHM

This section details the *FlexSample* packet selection algorithm that meets the two constraints mentioned in Section 2.3.

4.1 Overview

When a packet arrives, *FlexSample* first performs a counter lookup; the output of this step is an estimate of the count of each tuple. The second step is determining the sampling class to which the tuple-counts on the current packet correspond. Each sampling class has an associated sampling budget that remains constant for a particular input specification.

After determining the sampling class and budget for the current packet, *FlexSample* must calculate the probability of sampling the packet. This probability depends on both the sampling budget of

```
class CBFArray:
    var Array of Counting Bloom Filters cbfarray
    var Integer current_cbf
    function Lookup(key):
        let count = count for key from cbfarray[current_cbf]
        call Increment(key)
        return count
    function Increment(key):
        for each cbf in cbfarray
            increment counters for key
        end for
    endfunction
    function Rotate():
        clear cbfarray[current_cbf]
        current_cbf = (current_cbf + 1) modulo (size of cbfarray)
    endfunction
end class
```

Figure 4: Pseudocode for operations on CBFArrays.

the packet, and the instantaneous fraction of traffic that packets of the sampling class account for. The instantaneous fraction of *future* traffic that each sampling class accounts for cannot be pre-computed, so *FlexSample* approximates these fractions using estimates based on past observations of traffic fractions. The past observations of traffic fractions for each sampling class are then combined with the latest observations using Exponential Weighted Moving Average (EWMA) (with higher weight given to later estimates) to obtain a prediction for the future traffic fractions of that sampling class.

4.2 Details

Figure 5 presents the *FlexSample* procedure (Table 2 defines notation). The *FlexSample* procedure maintains four data structures: (1) m CBFArray counters, one corresponding to each tuple being counted. These are queried for each packet, and the corresponding counts, $(c_1, c_2, \ldots, c_m)$, are returned; (2) An m-dimensional array (similar to the sampling budget lookup table) to maintain the running estimate of traffic fractions occupied by packets belonging to each sampling class (f); (3) An m-D array to maintain the instantaneous sampling probabilities for each sampling class (γ); (4) An m-D array to maintain the counts of packets belonging to each sampling class in *actual* traffic (packets_per_sc). Finally, the integer packet_count maintains a running counts of all packets seen, in order to periodically trigger counter rotation.

Calculating the instantaneous sampling probability for each sampling class is the key operation in the procedure. The insight in calculating the instantaneous sampling probabilities is the following: *if the fractions of packets seen on the wire corresponding to a sampling class i, f_i, differs from the specified sampling budget for packets in that sampling class (α_i), then the overall sampling rate, s, must be scaled by the fraction $\frac{\alpha_i}{f_i}$.*

Two operations are performed periodically: rotating counters and re-calculating instantaneous sampling probabilities. Each period is called an *epoch*, which can be measured either in terms of total packets seen or in terms of time elapsed. At the end of an epoch, *FlexSample* performs the following maintenance operations.

(1) Rotating counters. CBFArray counters are rotated to age out stale counter values. The epoch lengths for each CBFArray are typically different, depending on the nature of the tuple being counted (*e.g.*, a tuple consisting of just the source IP address will likely have fewer entries than a tuple consisting of $(srcip, srcport, dstip, dstport)$); the presented algorithm rotates all CBFArrays together, but CBFArrays may also be rotated individually.

Notation	*Definition*
s	The original sampling rate (*i.e.*, for naïve random sampling).
m	The number of tuples (also called variables) in the specification
$v_1, ..., v_m$	Ordered set of the m tuples
$\|v_i\|$	The number of exhaustive, disjoint ranges for v_i
$T_{(v_i,1)}, ..., T_{(v_i,\|v_i\|+1)}$	The set of disjoint ranges for v_i. The first element ($T_{v_i,1}$) is always 0 and the last element ($T_{v_i,\|v_i\|+1}$) is always ∞.
CBFArray$[i]$	The CBFArray structure that is used for counting tuple v_i
sc	An abstract m-dimensional structure of size $\prod_{i=1}^{m} \|v_i\|$, representing the mutually exclusive, exhaustive set of sampling classes
$sc[c_1, \ldots, c_m]$	The sampling class to which a packet, whose instantaneous tuple-counts are $(c_1, c_2, \ldots, c_m)$, belongs. Its entries range from $[0, \prod_{i=1}^{m} \|v_i\|]$.
α	The m-dimensional sampling budget lookup table of size $\prod_{k=1}^{m} \|v_i\|$ (*i.e.*, $\|sc\|$)
α_i	The sampling budget for a packet that belongs to sampling class i
f	The m-dimensional traffic fraction lookup table of size $\prod_{i=1}^{m} \|v_i\|$ (*i.e.*, $\|sc\|$)
f_i	The instantaneous estimate of the traffic fraction of the sampling class i.
γ_i	The *instantaneous* sampling probability for sampling class i.

Table 2: Summary of notation.

(2) Re-estimating instantaneous sampling probability. Because the traffic mix in the input packet stream is dynamic, the traffic fractions of packets belonging to each sampling class may change. Thus, *FlexSample* must periodically recompute the traffic fractions of each sampling class; consequently, it must also recalculate the instantaneous sampling probabilities for each class. While recomputing traffic fractions, *FlexSample* retains some amount of memory for past fractions such that short-lived changes in traffic mix does not cause erratic updates to sampling probabilities, which could result in an undesired mix of sampled packets. To obtain the new table of traffic fractions, *FlexSample* uses an Exponential Weighted Moving Average (EWMA) on the latest and cumulative past estimates of traffic fractions, with higher priority typically given to the latest estimates. The intervals when *FlexSample* rotates counters and recalculates sampling probabilities need not be the same; in Figure 5, the operations are performed with the same periodicity for simplicity.

5. IMPLEMENTATION

We implemented *FlexSample* in approximately 3,000 lines of C++ code. We discuss design decisions for the counter implementation in Section 5.1, and for lookup tables in Section 5.2.

5.1 Counter Implementation

CBFArray, *FlexSample*'s counter structure, trades off space for efficiency. CBFArrays consume more space than necessary because there exist multiple copies of the same key among the elements in the CBFArray (*FlexSample* performs inserts to all CBFs in a CBFArray), but staggering multiple CBFs and periodically rotating through them expires inactive entries. This approach is more efficient than a timer-based approach. First, timer interrupts over

```
procedure FlexSample:
// Using notation from Table 2
Input:
    Packet stream; $P$
    Set of the tuples to count; $F$
    Original sampling rate; $s$
    Structure representing sampling classes; $sc$
    Sampling budget lookup table; $\alpha$
    Maximum packets per epoch; pkts_per_epoch
Output:
    Sampled Packets, the sampling class of each packet,
            and the instantaneous sampling probability of each packet
Algorithm:
// Array of CBFArray counters, one for
// each input variable being counted
counters := new 1-D ARRAY of type CBFARRAY of size $m$
// Structure representing instantaneous traffic
// fractions for each sampling class
$f$ := new $m$-D ARRAY
// Structure representing instantaneous sampling
// probabilities for each sampling class
$\gamma$ := new $m$-D ARRAY
// Structure that counts the number of packets
// in an epoch that mapped to each sampling class
packets_per_sc := new $m$-D ARRAY
// To count packets in each epoch
packet_count := new INTEGER
// Initially assume equal traffic fractions for all sampling classes
set each entry of $f$ to $1/|sc|$,
            the number of sampling classes
for each sampling class $i$ in $sc$
    set $\gamma_i = (\alpha_i \times s)/f_i$
end for
set each entry of packets_per_sc to 0
for each packet $p$ from stream $P$:
    if packet_count++ $>$ pkts_per_epoch
        call ROTATE for each counter in counters
        for each sampling class $i$ in $sc$
            // Recalculate $f$ using EWMA of current and past estimates
            $f_i$ = EWMA ($f_i, packets\_per\_sc_i/packet\_count$)
            // Recalculate $\gamma$ using new values of $f$
            $\gamma_i = (\alpha_i \times s)/f_i$
        end for
        set each entry of packets_per_sc to 0
        set packet_count to 0
    end if
    let key[1..$m$] = keys constructed by concatenating
        each set of fields in $F$
    let $[c_1, ..c_m]$ = counts of each key, by looking up each key
        key[1..$m$] in counters[1..$m$], in that order
    let class = $sc[c_1, c_2, .., c_m]$
    increment $packets\_per\_sc_{class}$
    sample packet $p$ at probability $\gamma_{class}$
end for
```

Figure 5: Definition of the *FlexSample* **procedure. The m-D arrays f, γ and** *packets_per_sc* **have the dimensions of the sampling budget lookup table α and are initialized to 0. The procedure** EWMA **accepts two floating point values and computes an exponential weighted moving average on the values using a pre-defined factor.**

100KHz can potentially result in up to 45% overhead for the processor merely in responding to interrupts [37]. 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 entry that has been inactive for a period longer than the time it takes to rotate all once through all CBFs in the array will exist in any of the CBFs. Correct choice of the number of CBFs and the CBF rotation condition (*i.e.*, a time interval or witnessing a certain number of packets) can reasonably approximate 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.*, during a DoS attack); therefore, we use this scheme in our implementation.

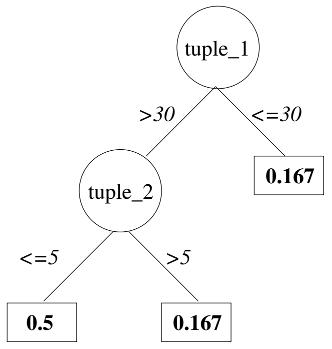


Figure 6: The sampling budget lookup table for Figure 2 represented as a binary decision diagram.

Using the target CBF error rate and total SRAM available, we can compute the overall number of entries the CBFArray can accommodate using standard Bloom Filter calculations [16]. The number of CBFs used in a CBFArray depends both on the nature of the tuple being counted and its dynamism: if the number of unique values the tuple can have is large, each CBF must accommodate a larger number of entries. Similarly, if the tuple's count is meaningful only in the short term, the CBFs in the CBFArray must be rotated quicker. Finally, the number of CBFs in a CBFarray decide the fraction of the tuple's count that is aged out at each rotation.

In our experiments, the default setting for a CBFArray uses 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. The amount of memory consumed increases linearly with respect to the number of entries the counter can accommodate. Because CBFArrays are rotated every epoch, we set the epoch size to the (number of entries)/(number of CBFs) packets, *i.e.*, 25,000 packets. Because sampling probabilities are updated at the end of each epoch, shorter epochs enable quicker updates and more traffic responsiveness. Longer epochs induce less updates but also consume fewer resources.

5.2 Lookup Table

In our implementation, the m-dimensional lookup table entries are represented as a linear array; thus, only the mapping of counter values to ranges is non-trivial. *FlexSample* uses binary search to quickly find the range corresponding to a tuple-count; thus, the complexity of a lookup is logarithmic in the number of ranges for a tuple. Lookups for each tuple count can be performed in parallel.

Maintaining multiple lookup tables also incurs memory overhead. The memory consumed by each lookup table is proportional to the product of the number of ranges for each tuple; thus, the memory consumption could increase if the operator adds more tuples or more ranges per tuple. However, each entry of a lookup table is typically 4 bytes, and even a specification that has 4 tuples each with 4 ranges consumes only about 1 kilobyte per table. Moreover, because a *FlexSample* specification typically reflects a small set of subpopulations the operator wishes to monitor, it is unlikely that a large number of tuples will be useful for a *single* specification, or that tuples will have a large number of distinct ranges.

If memory is a bottleneck, *FlexSample* can eliminate lookup tables altogether and instead store only the conditions from the input specification. In this case, the conditions and sampling budgets would be stored in a balanced binary decision diagram (BDD). Figure 6 shows a BDD representation of the sampling budget lookup table for Figure 2. Each internal node of the BDD would contain a condition, and each leaf of the diagram would contain a sampling budget. Using a balanced BDD, memory use is limited to the number of distinct conditions, but lookup time increases from constant order to logarithmic in the number of conditions.

6. APPLICATIONS

We evaluate *FlexSample* in the context of three applications. Table 3 lists these applications and the subpopulation definitions we use for these applications. Section 6.4 presents other applications that can use *FlexSample* as is or with little modification.

We compare *FlexSample*'s performance with naïve uniform random sampling (used by the popular Cisco Sampled NetFlow framework) as the baseline. In all cases, *FlexSample* meets sampling budgets for each class as well as the overall sampling rate[2].

Subpopulation	Applications	Evaluation
Very low-volume flows, containing less than 10 packets	(1) Identifying number of unique flows / sources, (2) Identifying email sending patterns	Sec. 6.1
Servers with high indegree of small flows	Identifying popular botnet command-and-control servers	Sec. 6.2
Sources sending a large number of packets to a destination, with only a few packets to any port	Identifying portscan attempts	Sec. 6.3

Table 3: Summary of the subpopulations we test *FlexSample* with, and their applications.

6.1 Identifying Low-volume Flows

The communication patterns between groups of hosts often reveal important information about the behavior of applications and their users. If these communications have many small-volume flows, naïve traffic sampling will fail to capture packets from these flows. In contrast, *FlexSample* can be used to capture a larger number of these flows by allocating a portion of the sampling budget (*e.g.*, 50%) to sampling packets from flows *that have not been observed previously in traffic*. Figure 7 shows an example input specification for this purpose[3]. The operator can tune the input condition in two ways: (1) *Change the range specification for the flow tuple:* increasing the range from `(0, 1]` to, say, `(0, 10]` will capture fewer unique flows but can estimate the actual size of flows with less error; (2) *Change the sampling budget allocated to low-volume flows:* a higher sampling budget for packets matching the condition will result in more unique flows in the sampled output.

[2]We omit this evaluation as it follows from the *FlexSample* sampling algorithm definition.

[3]Allocating 50% of the sampling budget to small "mouse" flows will capture more packets than naïve sampling only if the fraction of mouse *packets* in the traffic is *less* than 50%.

Name	Description	Period
Botnet	Packet captures of C&C traffic of the Bobax botnet	November 22, 2005
Email	All traffic destined to port 25 (SMTP) from a campus border router	7:30–10 AM, April 4, 2006
CoC_1	Full packet captures from a department network with over 3000 unique hosts	12:40–1:40 PM, April 15, 2008
CoC_2	Full packet captures from the department network containing a emulated portscan	12:40–1:40 PM, April 15, 2008

Table 4: Description of the various data traces we use in our evaluation.

```
sampling_rate = 0.01
tuples = 1
conditions = 1
tuple_1 := srcip.srcport.dstip.dstport.protocol
tuple_1 in (0, 1] :  0.5
```

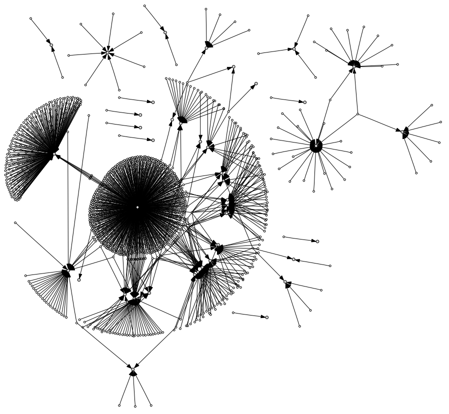
Figure 7: *FlexSample* input specification file for recovering unique flows. 50% sampling budget is allocated to flows which haven't been seen before.

Section 6.1.1 evaluates *FlexSample*'s ability to capture low-volume flows, and Section 6.1.2 shows how email sending patterns obscured by naïve sampling can be preserved with *FlexSample*.

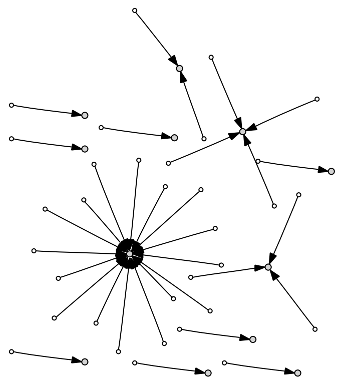
6.1.1 Capturing unique flows

We evaluated the specification in Figure 7 on the CoC_1 trace (Table 4), both for naïve sampling and *FlexSample*. Figure 8 plots the fraction of unique flows captured by *FlexSample* and naïve sampling. *FlexSample* captures fewer unique flows than naïve sampling with a sampling budget less than 0.4 allocated to the condition, because a large fraction (over 70%) of flows in CoC_1 have exactly 1 packet. In this case, the instantaneous sampling probability allocated to these packets would be *less* than the fixed 0.01 sampling rate used by naïve sampling. *FlexSample* steadily captures more unique flows on increasing the sampling budget over 0.4, with a budget of 0.9 capturing 46% more unique flows than naïve sampling. Moreover, the improvement over naïve sampling for the subpopulation of one-packet flows was almost 2.4 times.

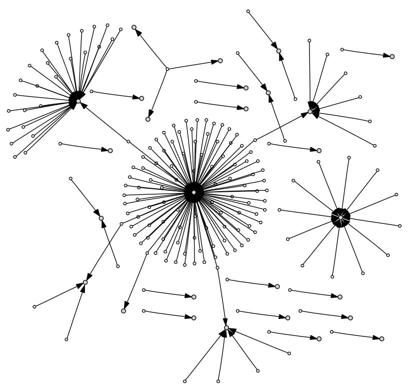
An improvement of only 46% over naïve sampling after allocating 90% sampling budget to low-volume flows may seem disproportionate, but this occurs because the total number of unique flows captured depends not just on the sampling budget, but also on the traffic flow size distribution: if a significant fraction of flows in the desired subpopulation are small-volume but have greater than 1 packet, allocating a budget of 0.9 to the condition in Figure 7 may elicit *less* flows than with a budget of, say, 0.5. The reason is that, with a 0.9 budget, if a flow of size greater than 1 is *not* selected at the first packet (at a high instantaneous sampling probability), all later packets in the flow will be sampled at a very low instantaneous sampling probability because only 10% of the sampling budget is allocated to such packets. Thus, maximizing the number of unique flows requires knowledge of the traffic mix; the optimal setting may not always correspond to the highest allocation of sampling budget to the first packet of each flow. We return to this issue in Section 8.4.



(a) Unsampled



(b) Naïve sampling



(c) *FlexSample*

Figure 9: Communication graphs of single-packet email flows. Arrows point from email senders to mail servers.

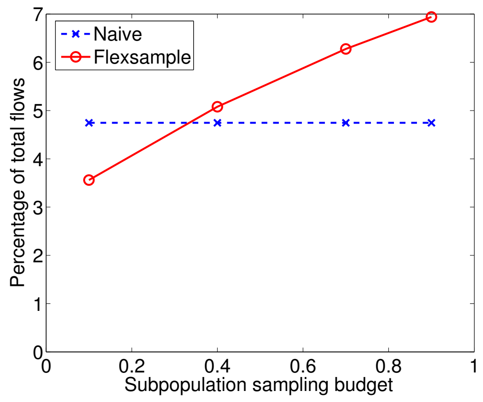


Figure 8: Effect of sampling budget on the number of unique flows captured with *FlexSample*. We compare the performance of naïve sampling with *FlexSample*.

6.1.2 Identifying email patterns

Spam and bulk email has surged over recent years, growing to over 90% of traffic [40]. Recent studies have shown that studying the sending patterns of email senders, such as domains they target, can help distinguish spammers from legitimate senders [32]. While this behavior can be gleaned from network traffic, naïve traffic sampling may often miss the behavior because SMTP transactions tend to be small-volume (recent studies indicate that the average filesize for spam is around 2 kilobytes [39], which would amount to very few packets on the wire).

To see how naïve traffic sampling affects email communication patterns, we use the Email trace (Table 4) and analyze the communication between mail senders (*i.e.*, client machines) and servers graphically. To visually discern the effect of sampling on the graph, we picked only those SMTP flows that consisted of only 1 packet—the flows that are the easiest to miss—and constructed an unsampled subgraph using these flows. Figure 9(a) shows this graph. Of 141,529 unique SMTP flows, 3,916 had just 1 packet.

Next, we performed naïve sampling at a rate of 0.01 on the one-packet SMTP flows. Of the 3,916, only 34 flows were in this set. Figure 9(b) shows this graph. Finally, we applied *FlexSample* on the Email trace with a sampling budget of 0.5. *FlexSample* sampled 224 out of 3,916 flows—almost 7 times the output of naïve sampling (Figure 9(c)). Comparing the graphs of naïve sampling and *FlexSample*, we see that *FlexSample* preserves information about the key email senders who send email to multiple servers.

6.2 Identifying High-Degree Nodes

Servers with high fan-in or clients with high fan-out are also of interest to an operator. Although popular servers that serve large volumes of content may be detectable even with naïve sampling, that technique may not be effective at identifying high in-degree servers whose flows are predominantly small-volume. For example, botnet "command-and-control" (C&C) servers are high in-degree nodes: they typically accept connections from many infected machines, but because the data exchanged is mostly short commands, these flows are usually small-volume. Detecting C&C communication is important to a network operator to take action against the C&C server or to enumerate potentially infected machines.

```
sampling_rate = 0.01
tuples = 2
conditions = 1
tuple_1 := dstip.dstport
tuple_2 := srcip.dstip.dstport
tuple_1 in (1000, ∞] AND
    tuple_2 in (0, 10]:  0.9
```

Figure 10: *FlexSample* input specification file for recovering unique flows from high-indegree servers. 90% of the sampling budget is allocated to the first 10 packets in a flow that hit a server which has already seen 1000 or more packets to the destination port on the current packet.

Although configuring *FlexSample* to select unique flows will work better than naïve sampling for capturing botnet flows, *FlexSample*'s configuration language can be used to tailor conditions more suitable to the high in-degree property that C&C servers exhibit. We use the specification in Figure 10 to preferentially pick the first few packets from flows that target an already popular destination server/port combination. While the first clause of the condition may match *any* popular server/port that has received over 1000 packets, the second clause only matches the first 10 packets of a *new* flow to the popular server/port.

To evaluate how well the condition captures unique flows to a botnet C&C server, we generated a synthetic trace that contains known botnet C&C traffic (the Botnet trace in Table 4) interspersed within regular traffic (CoC_1 from Table 4). We had to mix the two

	Botnet Packets	Botnet Flows
Total	303,238	78,356
Naïve sampling	3,108	3,030
FlexSample (using Figure 7)	3,592	3,471
FlexSample (using Figure 10)	12,960	11,885

Table 5: Summary of botnet traces and results of evaluation with various sampling methods. Base sampling rate was fixed at 0.01 for all.

traces because we do not have packet traces for all traffic for the duration of the botnet trace (or vice versa). When mixing Botnet with CoC_1, we used botnet data from the same day-of-week and time-of-day as CoC_1 to minimize biases. We also marked the botnet C&C packets in the mixed trace to allow us to identify them later. Table 5 summarizes the number of botnet packets and the number of unique botnet flows in the mixed trace, and the effects of naïve sampling, *FlexSample* using the specification in Figure 7 (90% budget allocated to the first packet), and *FlexSample* using the specification of Figure 10 on these metrics. The high in-degree specification outperforms both naïve sampling and *FlexSample* configured to select only unique flows: it captures almost one-seventh of all botnet flows.

6.3 Capturing Portscan Packets

In this section, we evaluate the portscan example presented earlier in the paper (Figure 2). Portscans are typical precursors to vulnerability exploits or DDoS attacks [33], and detecting them quickly is crucial. Portscans come in many varieties, such as SYN scans (scanning using SYN packets expecting SYN+ACK for listening servers, or RST packets in return), FIN scans (scanning using FIN packets expecting RST for non-listeners and no response for listeners), and advanced techniques such as Idle scans [1] or FTP bounce scans [2]. In all these cases, the key artifact of the scan is a source that probes many ports on a destination using few packets per destination port.

Due to lack of traces with real portscans, we conducted our own portscan of a /24 subnet using the popular *nmap* tool [38]. We used the TCP SYN "stealth" scan mode for our test that randomizes the order in which target ports are scanned, and sends very few packets per second so as to not trigger alerts; this type scan would be harder to detect using simple volume-based or sequential port number heuristics. The trace containing the portscan is marked CoC_2 in Table 4. CoC_2 had over 116 million packets, of which only 0.044% were portscan packets. We then evaluated the performance of naïve sampling and *FlexSample* (using specification in Figure 2) in extracting portscan packets in sampled traffic. As Figure 11 shows, even at low budgets, *FlexSample* captures more portscan packets than naïve sampling; this fraction increases to 50% of portscan packets using a high sampling budget, even for a 1-in-100 overall sampling rate.

6.4 Other Applications

We briefly discuss a few additional applications of *FlexSample*

DDoS Detection Using Packet Lengths. *FlexSample* can use the IP packet length field in tuple specifications to capture Distributed Denial of Service (DDoS) packets. DDoS attacks are usually performed by bots in the same botnet (running the same codebase), where each bot is performing a type of flooding attack (*e.g.*, TCP SYN flood, UDP Flood, HTTP request flood, etc.). Because flooding attacks correspond to sending many packets with the same payload, packet lengths may be the same for all DDoS packets. A

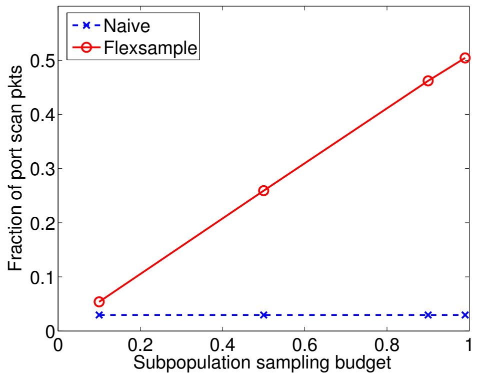


Figure 11: Comparison of *FlexSample* and naïve sampling on portscan packet extraction, with varying sampling budgets allocated to the portscan condition.

specification that exploits this artifact of DDoS attacks is shown in Figure 12. This specification allocates greater sampling budget for packets from sources that: (1) are hitting a very popular TCP server/port, (2) have sent at least 500 packets of the same size to the server/port.

```
sampling_rate = 0.01
tuples = 2
conditions = 1
tuple_1  := dstip.dstport.tcpsyn
tuple_2  := srcip.dstip.dstport.pktlen
tuple_1 in (10000, ∞] AND
    tuple_2 in (500, ∞] :  0.5
```

Figure 12: *FlexSample* input specification file for preferentially sampling DDoS packets exploiting the commonness of packet length among DDoS packets.

Identifying Heavy-Hit IP Ranges. Many popular servers (*e.g.*, Web servers such as `myspace.com`) are load-balanced using multiple IPs within the same netblock (usually a /24). In such setups, each individual server may not be popular enough on its own, while together, they account for a significant fraction of flows. An operator can identify such popular netblocks by applying a simple modification to the high-degree server identification: instead of $dstip$ in the specification of Figure 10, he can use the first 24 bits of $dstip$, which is common to all load-balanced servers in a /24 netblock.

7. FLOW SIZE ESTIMATION ACCURACY

Although network operators may wish to monitor small-volume subpopulations, their primary use for sampled traffic is to estimate the flow sizes of heavy-hitter "elephant" flows accurately [10, 17]. Thus, *FlexSample* must not incur such high error in flow size estimation of elephant flows over naïve sampling such that operators' primary objective may be compromised.

Estimating actual flow-sizes for naïve sampling involves scaling up the number of packets per flow in sampled output using the fixed base sampling rate. In contrast, because *FlexSample* uses different sampling probabilities for each packet, we also output, with each sampled packet, the probability at which it was sampled. Based on this probability, we can compute an estimate of the actual flow size. Our approach in computing flow size estimate is similar in spirit to [26]. We compute the flow size estimate, $\widehat{S}(f)$, of flow f

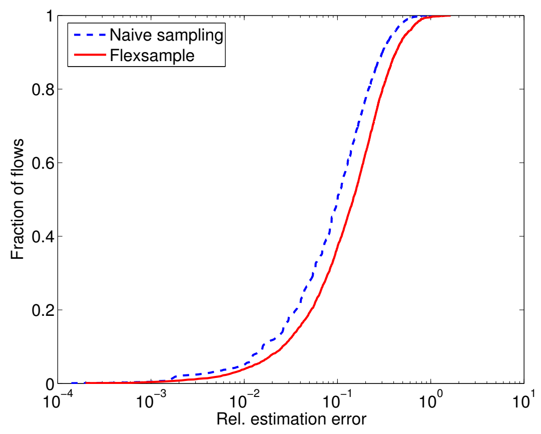


Figure 13: CDF of the absolute value of the relative estimation error incurred by individual flows with *FlexSample* and with naïve sampling. With *FlexSample*, 70% of the sampling budget is allocated to the first packet of each flow.

with $n(f)$ packets, where packet i was sampled at a rate $r(i)$ as

$$\widehat{S}(f) = \left\lfloor \sum_{i=1}^{n(f)} \frac{1}{r(i)} \right\rfloor$$

Flow size estimation incurs some estimation error in extrapolating the sampled flow size to the original flow size. For naïve sampling, this error for a flow f of actual size $S(f)$ is

$$\text{Error }(f) = \frac{S(f) - \widehat{S}(f)}{S(f)}$$

Figure 13 shows the cumulative distribution of the absolute values of relative estimation error incurred per flow, both for *FlexSample* and naïve sampling, using the CoC_1 trace. The *FlexSample* experiment used the specification of Figure 7, with 70% of the total sampling budget allocated to small-volume flows. We plot the estimation error for flows of size greater than 1,000 packets, as they contribute to over 40% of the overall traffic volume. The estimation error incurred for these flows is almost the same for both *FlexSample* and naïve sampling. For 80% of these flows, the increase in estimation error incurred by *FlexSample*, in comparison to naïve sampling, is less than 9%. Even when 90% of the sampling budget is allocated to small-volume flows, we see that the increase in estimation error incurred by *FlexSample*, for 80% of the flows, is within 30%. This shows that *FlexSample* captures more unique flows with modest increase in estimation error for heavy-hitter flows sizes.

8. DISCUSSION

In this section, we discuss practical issues in deploying *FlexSample* (Section 8.1 and Section 8.2), and how *FlexSample* may be configured to emulate "smart" sampling algorithms such as *sample-and-hold* [15] (Section 8.3). Section 8.4 discusses how traffic dynamism affects *FlexSample*, and how *FlexSample* can *automatically* compute sampling budgets to mitigate its traffic dependence.

8.1 Practical Issues

We discuss four issues: reducing noise in sampled packets, increasing the portability of specifications using join variables, clauses using static matches, and an extension to *FlexSample* that allows network-wide coordination between different *FlexSample* instances.

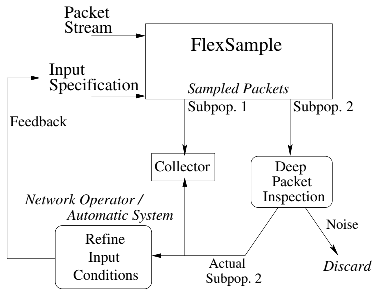


Figure 14: To reduce noise in sampled packets of Subpopulation 2, an operator can tune the conditions and tuple ranges in the input specification using feedback from a deep packet inspection device.

a. Reducing noise in sampled packets. As our evaluation in Section 6.1.1 showed, conditions based on tuple counts have inherent error: (1) some packets of a desired subpopulation may *not* match the condition for the subpopulation, and (2) packets of *other* subpopulations ("noise") may match the condition and waste its budget. *FlexSample* cannot automatically filter this noise because it observes a packet only in terms of tuple counts; typically, the sampled packet's *contents* must be inspected to determine whether it belongs to the sought subpopulation.

To reduce noise within sampled packets of a subpopulation, the operator must refine the input specification (*e.g.*, by tweaking the ranges of tuple counts, adding new conditions, etc.). To determine the tweaks, he could inspect a full packet trace to identify the ranges of tuples that best distinguish the desired subpopulation from other packets. However, obtaining a full packet trace on a high-speed link may not be feasible (indeed, this is why operators must resort to traffic sampling). Alternatively, the operator can use a system as in Figure 14: the operator shunts packets sampled as Subpopulation 2 through a deep packet inspection (DPI) device that authoritatively identifies packets that actually belong to Subpopulation 2. The operator can then refine input conditions—either manually or using an automatic system—for the next epoch. Because DPI is performed only on a portion of sampled traffic, its complexity does not affect the rest of the sampling process.

b. Join variables. The problem of deciding correct ranges for tuples is exacerbated in practice because the tuple ranges that work well to capture a particular subpopulation (*e.g.*, portscan packets) on one network may include many false positives when used on another network. To mitigate this problem, the *FlexSample* configuration can be modified to use variables instead of fixed numbers for ranges. The variables act as "join" keys between various clauses in a condition. For example, Figure 15 shows the condition of Figure 12 using the join variable `p`: the condition now specifies that a destination receives `p` connections, out of which a significant fraction is due to equal-length packets originating from the source IP on the packet currently under inspection. The shown condition places no lowerbound on the value of p; if the lack of a lowerbound causes false positives, a clause such as `p in (500,` ∞`]` may also be appended to the condition.

c. Static Conditions. Although the power of the *FlexSample* language is in capturing subpopulations based on *behavior*, static matches can also be used as clauses. For example, if the operator knows that DDoS sources are targeting only Web servers, he

```
  tuple_1 in (p, ∞] AND
      tuple_2 in ((p/20), ∞] :  0.5
```

Figure 15: The condition of Figure 12 updated to include a join variable p.

can add an extra clause to the condition of Figure 12 such that the condition will match only if $dstport$ is equal to 80.

d. Network-Wide Monitoring. cSamp [35] takes a network-wide view of sampling: it uses uniform hashing functions on flow tuples on all routers in a network but requires each router to sample flows based on disjoint *hash ranges*. Thus, in cSamp, multiple routers that see the same flow will not duplicate sampling effort, without explicit communication. Although *FlexSample* is primarily a packet selection technique from a single router's perspective, because *FlexSample* uses the hashed counters, it can also benefit from a scheme such as cSamp. For example, if each router's counters *only* count the tuples that hash to disjoint ranges, different routers can monitor the same subpopulation without sampling duplicates.

8.2 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 arguments as to why both requirements are reasonable, at least for today's high-end routers and network devices.

SRAM limits. SRAM on routers can be on-chip or off-chip. According to Varghese ([37, page 441]), on-chip SRAM has latencies below 5ns, and is limited to about 64 megabits. Off-chip SRAM can provide higher capacities at slightly higher latencies. In our experiments, the maximum memory used by the CBFArray structure was less than 6 MB, 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 [34]. 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 3 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. There are two issues to consider: (1) How feasible is hashing every packet, and (2) If not every packet is counted, how much does the accuracy of estimating the counts of a tuple *degrade*?

Fast Hashing. 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 [36], 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 [42].

Hashing fewer packets. *FlexSample* counts tuples by computing hashes over keys formed by concatenating tuple fields from the packet. 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) [4]. k is set to 1 in our experiments. Fast hashes over well-defined bytes of a packet can be performed at very high speeds [42] (making hashing every packet practical), but even if the counter cannot examine every packet, we present an argument in Appendix A. that the classifier should still perform well.

8.3 Emulating Other Sampling Algorithms

Estan *et al.* proposed *sample-and-hold* as a method to provide increased estimation accuracy for high-volume flows over naïve sampling [15]. Their algorithm functions similar to naïve sampling at a fixed probability, but whenever a packet of a certain flow has been sampled, *all* further packets from the flow are selected *deterministically* (*i.e.*, no sampling). To do so, they use in-memory data structures similar to *FlexSample*.

```
  sampling_rate = 1.0
  tuples = 1
  conditions = 1
  tuple_1 := srcip.srcport.dstip.dstport.protocol
  tuple_1 in (1, ∞] :  1.0
```

(a) Specification 1

```
  sampling_rate = 0.01
  tuples = 1
  conditions = 1
  tuple_1 := srcip.srcport.dstip.dstport.protocol
  tuple_1 in (0, ∞] :  1.0
```

(b) Specification 2

Figure 16: *FlexSample* input specification files that emulate the *sample-and-hold* algorithm.

Because of the additional "always-pick" condition, total number of packets selected by *sample-and-hold* may often exceed the total number of packets selected by naïve sampling. Because *FlexSample* never exceeds the allotted sampling budget, a single instance of *FlexSample* will not be able to emulate *sample-and-hold*. We can, however, emulate *sample-and-hold* using *two FlexSample* instances, each with different specifications. Figure 16 shows these specifications. The two *FlexSample* instances are chained using a condition: Specification 1 is applied first to each packet, and Specification 2 is applied on the packet only if Specification 1 is not matched (*i.e.*, if the packet was not selected using Specification 1). Because the two specifications share the same counter, there is no extra memory overhead.

These examples show that *FlexSample*'s configuration language is powerful and can be adapted to meet a variety of requirements.

8.4 Choice of Sampling Budgets

Operators ultimately want to sample enough packets of each subpopulation to obtain a representative sample of the subpopulation.

[4]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.

This goal has two requirements: (1) The condition(s) specifying a subpopulation must match a large fraction of packets of that subpopulation and few packets of other subpopulations (*i.e.*, few false positives *and* false negatives); (2) The sampling budgets chosen for each subpopulation must neither be too high nor too low.

We already discussed the first requirement in Section 8.1(a). The second requirement is also important: if the desired subpopulation occupies a small fraction (*e.g.*, 0.01) of all traffic, even a high sampling budget (*e.g.*, 0.9) may not capture enough packets to fill the budget. Thus, the budget of 0.9 will never be completely used leading to wasted budget (for other subpopulations). In our current implementation, operators have to guess static budgets *a priori*, but optimal budget choice requires knowledge of the *actual* (*i.e.*, future) fractions of the subpopulation, and dynamic adjustment of sampling budget based on these fractions. In spite of this limitation, in our evaluation, we were able to capture enough packets from desired subpopulations with small changes to the conditions or budgets; we believe an operator would be able to arrive at the correct settings in a few tries as well.

In future work, we plan to explore the following alternatives: (1) *Always Select k:* Instead of using an instantaneous sampling probability to select each packet, we always pick the first k_i packets from each subpopulation i, where k_i is the expected number of sampled packets that belong to subpopulation i (*i.e.*, subpopulation i's per-epoch sampling budget). For the remaining packets of subpopulation i, we select a packet with a probability inversely proportional to the index of the packet, and we randomly replace one of the chosen k_i packets with the selected packet. This scheme ensures that even if the fraction of packets belonging to a subpopulation are much lower than the budget for that subpopulation, all such packets will be deterministically selected. Others have also found that deterministically selecting the first few packets of a connection typically retains the most interesting part of the connection [30]. (2) *Dynamic Sampling-Budget Scaling:* Instead of rigidly fixing sampling budgets for each subpopulation at the first epoch, we will allow sampling budgets to vary (within upper and lower bounds) such that unused (or unneeded) budgets for some subpopulations can be allocated to subpopulations that need larger budgets. For example, if a subpopulation occupies 99% of traffic and has 50% of the budget, decreasing its budget to 45% will likely *not* affect the statistical significance of sampled packets of the subpopulation, while the freed 5% budget can be used to improve packet selection for a smaller subpopulation. Dynamic Sampling-budget Scaling can also be used in coordination with the Always Select k scheme.

9. RELATED WORK

FlexSample draws on a large body of previous work in fast statistics counters and various types of packet sampling techniques. We survey common packet and flow sampling techniques in Section 9.1, and review previous work on traffic monitoring applications (such as anomaly detection and traffic classification) and frameworks (such as ProgME [43]) in Section 9.2.

9.1 Sampling Techniques

Random Sampling. Traffic monitoring on high-speed links is typically performed with techniques such as Cisco's NetFlow [31], Juniper traffic sampling [21], or InMon sFlow [20]. Because these techniques incur both high processing and collection overhead, routers typically employ *packet sampling* on high-speed links. Sampled schemes (*e.g.*, Sampled NetFlow) observe only a fraction of all packets that traverse the link; summaries it exports reflect only the statistics of the sampled traffic.

Packet sampling inspects every nth packet—either deterministically or at random—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—a variant of random sampling—divides traffic into equal-length strata and selects packets randomly from within the strata at a particular sampling rate [46]; this approach resembles *FlexSample*'s division of traffic into subpopulations.

Flow sampling [18], in contrast with packet sampling, selects flows with some probability and retains all packets from selected flows. While flow sampling avoids the bias against mice flows observed with packet sampling, it is resource-intensive because all packets have to be assigned to flows before a sampling decision is made; hence, it is difficult to deploy in high-speed routers.

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 size [12, 13].

FlexSample's packet selection scheme is similar to *sketch-guided sampling* (SGS) [26], which samples packets with a probability distribution that depends on the size of the flow to which the packet belongs. That work develops a general theory for how such sketches might be constructed and serves as the basis for the high-level *FlexSample* design. *FlexSample* extends SGS by showing how such an approach can help network operators capture specific subsets of the traffic distribution *on the fly* with a simple set of tunable parameters. In particular, the sketch-guided sampling proposal focused on a different class of problems (*e.g.*, tracking elephants) and did not provide tunable parameters for capturing *post facto* fractions of low-volume subpopulations.

In an earlier work, Kumar *et al.* focused on combining flow records of sampled traffic with an approximate counts of traffic streaming through the observation points to generate estimates of flow statistics [25]. They extend this estimation to target only a subpopulation of the total flows, using value-based filters. Their motivation is to accurately infer flow sizes from previously generated flow records of sampled traffic. *FlexSample*, on the other hand, is concerned with online biasing of packet sampling.

9.2 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 and time-based granularities [8]. 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 [7, 11, 14, 15, 18, 23, 24, 44]. Previous research has also investigated how to achieve effective coordination across multiple traffic monitors to improve network-wide flow monitoring [6, 35]. These techniques adapt the sampling rate to changes in flow characteristics, attempt a different sampling strategy altogether, or apply network-wide constraints, typically to draw inferences about flow size distributions from sampled traffic statistics. In contrast, *FlexSample* uses a statistics counter to *control* the sampling process itself to solve a broader class of problems (*e.g.*, monitoring botnets, recovery of traffic structure, etc.).

Although we present a novel counter architecture and sampling algorithm in this paper, *FlexSample* is not tied to either. Thus, operators can gain advantage from recent advances in efficient counter architectures (*e.g.*, Counter Braids [27]), or choose to use sampling algorithms that respect constraints other than the ones described in this paper (*e.g.*, Adaptive Netflow [14] for surviving DoS attacks).

Application tracking and anomaly detection. Sampled traffic statistics have also been used to help operators detect malicious traffic [3, 4, 41], and many previous studies have demonstrated the utility of using sampled flow statistics for detecting high-volume attacks and malicious traffic [3, 14, 19, 45]. However, more recent work has demonstrated that conventional sampling techniques can obscure statistics needed to detect traffic anomalies [5] or execute certain anomaly detection algorithms [29]. Traffic classification studies have also used network communication structure to identify attack traffic [22, 41]. *FlexSample* can further assist these techniques by allowing them to focus on specific flow size ranges (*e.g.*, mouse flows).

Monitoring Frameworks and Languages. ProgME [43], an application-oriented flow monitoring framework, operates on arbitrary compositions of flows (called *flowsets*) that can be logically composed using a powerful composition language. ProgME's notion of a flowset is a generalization of *FlexSample*'s tuple concept, but *FlexSample*'s advantage is in showing how tuples can be combined using conditions that reflect a specific subpopulation's characteristics, in order to bias monitoring in the subpopulation's favor. PSAMP [9] is an IETF working group that is concerned with drafting a framework for allowing configurability in selecting, summarizing, and exporting statistics. Madhyastha *et al.* have constructed a language for efficiently filtering flow records generated by a flow collector [28]. While their work uses templates for extracting flow statistics *from sampled traffic*, *FlexSample*'s specification language is used to drive the packet sampling process itself.

10. CONCLUSION

Most existing high-speed traffic monitoring techniques allow operators to perform conventional tasks, such as tracking "heavy hitter" flows for traffic engineering or billing. Given evolving threats and applications, however, network operators have an increasing need to sample specific traffic subpopulations based on a flexible specification. Towards this goal, this paper has presented the design, implementation, and evaluation of *FlexSample*, a new framework and system for monitoring traffic that allows a network operator to skew sampling rates towards a particular range of the traffic distribution. *FlexSample* allows an operator to extract traffic subpopulations based on conjunctions of conditions that can be specified in terms of packet header fields, subject to an overall sampling constraint.

In this paper, we offer three new contributions. First, we present a new framework that decouples counting from sampling and uses a small amount of fast memory to guide the sampling of traffic from specific subpopulations. Second, we present a simple, expressive configuration language that allows operators to specify these subpopulations. Finally, we present the design and implementation of *FlexSample* and its use for several applications (capturing unique flows, high-degree nodes, and portscans). In each case, we show that *FlexSample* can capture significantly more of each of these subpopulations, at the cost of an acceptable increase in error rates for "conventional" traffic monitoring applications such as estimating traffic flow size distributions.

Acknowledgements

This work was supported by an NSF CAREER award CNS-0633974, and by NSF Cybertrust awards NSF CNS-0721581 and NSF CNS-0831841. We thank Russ Clark and David Dagon for the traces used in our analysis, and Anukool Lakhina for many inspiring discussions that laid the groundwork for this paper. We also thank the anonymous reviewers and Muhammad Mukarram Bin Tariq for comments that helped improve the paper.

REFERENCES

[1] Idle-scanning and Related IPID Games. `http://nmap.org/idlescan.html`.

[2] Original posting describing FTP bounce scan. `http://nmap.org/hobbit.ftpbounce.txt`.

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

[4] 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.

[5] 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.

[6] G. Cantieni, G. Iannaccone, P. Thiran, C. Barakat, and C. Diot. Reformulating the monitor placement problem: Optimal network-wide sampling. Intel Research Technical Report, Feb. 2006.

[7] 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.

[8] 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.

[9] N. Duffield. A Framework for Packet Selection and Reporting. IETF Internet Draft draft-ietf-psamp-framework-12.txt, June 2007.

[10] N. Duffield, C. Lund, and M. Thorup. Charging from Sampled Network Usage. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, San Fransisco, CA, Nov. 2001.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *IEEE/ACM Transactions on Networking*, 9(3):257–270, June 2001.

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

[19] 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.

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

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

[22] 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.

[23] R. Kompella and C. Estan. The Power of Slicing in Internet Flow

Measurement. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[24] 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.

[25] A. Kumar, M. Sung, J. Xu, J. Wang, and E. W. Zegura. A Data Streaming Algorithm for Estimating Subpopulation Flow Size Distribution. In *Proc. ACM SIGMETRICS*, Banff, Canada, June 2005.

[26] 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.

[27] Y. Lu, S. Dharmapurikar, A. K. Kabbani, A. Montanari, and B. Prabhakar. Counter Braids: An Efficient Minimum-Space Statistics Counter Architecture. In *To appear in the Proceedings of ACM SIGMETRICS*, June 2008.

[28] H. V. Madhyastha and B. Krishnamurthy. A Generic Language for Application-Specific Flow Sampling. *ACM Computer Communication Review*, 38(2), April 2008.

[29] 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.

[30] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching Network Security Analysis with Time Travel. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

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

[32] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM*, Pisa, Italy, Sept. 2006. An earlier version appeared as Georgia Tech TR GT-CSS-2006-001.

[33] H. Ringberg, A. Soule, and M. Caeser. Behavior Of Bots In Traffic Traces. Technical report, Princeton University, 2008. Number forthcoming.

[34] 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.

[35] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSamp: A system for network-wide flow monitoring. In *Proc. 5th USENIX NSDI*, San Francisco, CA, Apr. 2008.

[36] 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.

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

[38] F. Vaskovich. Nmap stealth port scanner. `http://www.insecure.org/nmap/index.html`, 2002.

[39] Average spam message size at record low. `http://www.virusbtn.com/news/2008/04_03a.xml?rss`.

[40] Report: 95 percent of all email has that spammy smell. http://arstechnica.com/news.ars/post/20071212-report-95-percent-of-all-e-mail-has-that-spammy-smell.html.

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

[42] 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.

[43] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards Programmable Network MEasurement. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[44] 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.

[45] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Taormina, Sicily, Italy, Oct. 2004.

[46] 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. HASHING FEWER PACKETS

We claim that *FlexSample*'s estimation of tuple-counts degrades gracefully even if tuples in every packet are not counted. Below, we provide a proof sketch for the argument, using a simple example that has only one tuple: the $(srcip, srcport, dstip, dstport, protocol)$ 5-tuple used to count the number of packets unique flows. We call the high-volume flows elephants, and the low volume ones, mice.

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 $\delta > 0$.

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 $e^{-2\mu\delta^2}$. ■