Per Flow Packet Sampling for High-Speed Network Monitoring

Marco Canini*, Damien Fay†, David J. Miller†, Andrew W. Moore†, Raffaele Bolla*
*DIST, University of Genoa, Italy
{first.last}@unige.it
†Computer Laboratory, University of Cambridge, UK
{first.last}@cl.cam.ac.uk

***Abstract*—We present a per-flow packet sampling method that enables the real-time classification of high-speed network traffic. Our method, based upon the partial sampling of each flow (i.e., performing sampling at only early stages in each flow's lifetime), provides a sufficient reduction in total traffic (e.g., a factor of five in packets, a factor of ten in bytes) as to allow practical implementations at one Gigabit/s, and, using limited hardware assistance, ten Gigabit/s.**

I. Introduction

Accurate real-time identification of network-based traffic is an important factor for solving difficult network management problems including network-security, accounting, traffic engineering, and new class-of-service offerings.

Traditionally, traffic classification was realised by simply inferring the controlling application's identity from the use of TCP or UDP port numbers under the assumption that most applications consistently use 'well-known' ports. However, this technique is no longer effective because many applications are increasingly using ephemeral port numbers [1].

Recent research on Internet traffic classification has brought many interesting ideas and methods that do not rely on 'well-known' port numbers. Most of these newer schemes classify traffic by recognizing statistical patterns in externally observable characteristics. Particularly, alongside Intrusion Detection Schemes (IDS) such as Snort [2], there have been a number of flow-classification mechanisms discussed in the literature (e.g., Li et al. [3], Bernaille et al. [4] and Crotti et al. [5]) that are able to provide high accuracy with access to only a limited number of packets from each flow.

However, in order to collect flow features and classify several hundreds of thousands of concurrent flows (which is typical of high-speed links), these schemes are associated with significant consumption of memory and computational resources. There are in fact definite trade-offs to be made between the classification performance and the resource consumption of the actual implementation [6].

In this paper we present a new method of inline, real-time, classification for high-speed networks. Our method, based upon the partial sampling of each flow, permits sufficient reduction in total traffic (e.g., a factor of five in packets, a factor of ten in bytes) to just that traffic required to permit practical implementations at one Gigabit/s, and, using limited hardware assistance, ten Gigabit/s.

There already exists a prodigious quantity of work on the sampling of streams of packets; for example, Duffield [7] provides an excellent review of the field. This research differs from most sampling techniques in that we specifically wish to target the first J packets of each flow within a traffic-multiplex. By focusing upon only a fixed number of packets early within each flow we can discard all remaining packets of any flow. Our approach provides the compliment to a number of existing network (application) classification schemes — permitting their realization at high speed.

We consider that our approach to flow-sampling permits the implementation of such traffic schemes for the future of advanced network monitoring, network resource management, anomaly detection, application-specific strategies and network auditing activities.

II. Methodology

This section gives an overview of the design of the per-flow packet sampling scheme. This scheme is shown diagrammatically in Figures 1 and 2. The scheme operates by defining a time window of length W and identifying the first J packets from each flow that occur within that window (typically several seconds long). As can be seen the scheme is structured into two levels. The first is the sampling mechanism which operates at the per-packet level (Figure 1) and the second is the memory allocation algorithm (Figure 2) which operates on a per-window level (and thus the time scale is several orders of magnitude higher) and sets the parameters required in the first level (specifically the m_j, $j = 1, \ldots, J$).

In this work, we define a flow as a bi-directional stream of packets identified by the usual IP five-tuple: source and destination addresses, IP protocol, and source and destination ports.

At the per-packet level every packet initially has its flow identifier extracted (i.e., the IP tuple). This identifier is then used to query the first Bloom filter to determine if it has been seen before (a Bloom filter may be considered a lossy memory device as will be explained in detail later in Section III). If it has been seen before then the second Bloom filter is queried

This work was done while Marco Canini was visiting the Computer Laboratory, University of Cambridge.

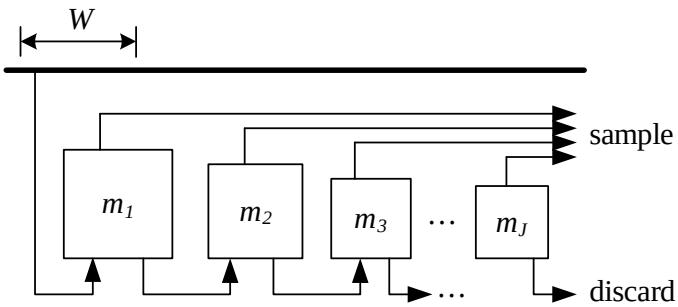


Fig. 1. Per-packet level of the sampling scheme.

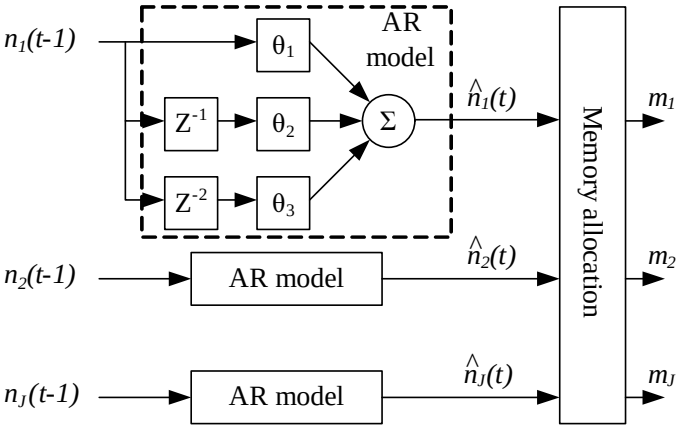


Fig. 2. Per-window level of the sampling scheme.

and so on until either a negative response is returned or all the Bloom filters return a positive response. A negative response in the jth filter indicates that this is the jth packet from this flow (in an ideal case) and it is thus selected for sampling. In addition, this IP tuple is added to the Bloom filter, so that the next packet from this flow will obtain a positive answer from this filter. In the case that all the Bloom filters return a positive answer then (ideally) this is not one of the first J packets in the flow and is discarded. Figure 3 shows the pseudo-code for the described algorithm. As will be explained later, the amount of IP tuples that the jth Bloom filter can hold is directly related to the amount of memory, m_j, assigned to this filter. The aim of the per-window level is to optimally divide a central block of memory, M, into J different portions. At the per-window level an estimate of the optimal memory allocation for the *next* window is required. As this is based on the number of IP tuples that will be stored in each Bloom filter, n_j, at the end of the *next* window a prediction of these values is needed. This is discussed in detail in Section IV.

III. Per-packet level sampling

A Bloom filter is a simple space-efficient probabilistic data structure for representing a subset $S = \{x_1, x_2, ..., x_n\}$ of n elements of some universe U in order to support membership queries. A Bloom filter is implemented as an array of m bits, initially all set to 0, uses k hash functions mapping elements

```
let P = current packet
let f = flow identifier of P
for i := 1 .. J
  b := BloomFilter[i]
  if query(b, f) = true
    if i = J
      discard(P)
      break
  else
    add(b, f)
    sample(P)
    break
```

Fig. 3. Algorithm for updating the Bloom filters within the time window.

in S to $[0, ..., m-1]$, and supports two basic operations: `add` and `query`.

The index functions are traditionally assumed to be hash functions with the standard assumptions that they are random, uniform, and independent, though these assumptions can be replaced with universal hashing arguments [8].

Given an element $x \in U$, a Bloom filter supports the operation `add(`x`)` which uses the hash functions to generate k indices into the array and sets the corresponding bits to 1.

The operation `query(`y`)` tests if an element $y \in U$ belongs to the set S by computing the k indices for y and checking whether all referenced bits are 1. A negative query clearly indicates that the element is not in the Bloom filter, but a positive query may be due to a false positive; the case in which the queried element was not added to the Bloom filter, but all k queried bits are one (due to other additions).

We now introduce the theory required for selecting the appropriate dimensions of the bloom filters. First a theoretical expression will be constructed and then progressively relaxed to allow easy online implementation. The theory behind Bloom filters is well known but is repeated here as this particular application has several deviations from the usual situation.

During an insertion, the probability that a certain bit is not set by a certain hash function is $1 - 1/m$. Considering all k hash functions, the probability that a certain bit is not set is:

$$P(\text{bit not set}) = \left(1 - \frac{1}{m}\right)^k \tag{1}$$

If i (distinct) elements have already been added then the probability that a particular bit is still 0 is:

$$P(\text{bit not set} \backslash i) = \left(1 - \frac{1}{m}\right)^{ki} \tag{2}$$

And so, when the filter contains i elements, the probability of a false positive (i.e., all k bits have been set) is:

$$P(\text{f.p.} \backslash i) = \left(1 - \left(1 - \frac{1}{m}\right)^{ki}\right)^k \tag{3}$$

It can be shown [9] that if a Bloom filter is designed to hold at most n (distinct) elements then the optimal value of k

is $\ln 2(m/n)$. This value minimizes the probability of a false positive after all n elements have been added to the filter. However, in this application we are interested in the number of false positives that occur as the Bloom filter is being filled. Consider the start of a new flow. Querying the first filter with the identifier from that flow *should* obtain a negative answer (if there is no false positive), thus the flow identifier will be included in the Bloom filter. Subsequent packets from this flow will *definitely* obtain a positive answer and so will be shunted onto the next Bloom filter for consideration (multiple filters will be considered later in this section). Thus, for analyzing the number of false positives, with respect to this Bloom filter, only the first packet from a new flow need to be considered. In fact, each first packet can cause exactly one false positive with probability given by Equation 3. Therefore, we can formulate the expected number of false positives for the first filter, $E[F_1]$, by summing over the total number of flows n, which gives:

$$E[F_1] = \sum_{i=1}^{n} \left(1 - \left(1 - \frac{1}{m}\right)^{ki}\right)^{k} \tag{4}$$

We now consider the case for multiple Bloom filters. Suppose there are J Bloom filters arranged such that any flow identifier that obtains a match in the first filter is tested for a match in the second and so on until either no match is found (and so this flow identifier is recorded in that filter and the current packet gets sampled) or all the filters report a match (and so the current packet is discarded). Assuming a flow with a number of packets greater than the number of filters, ideally the jth packet from this flow should cause a match in all the filters between the first up to the $j-1$th, while the jth filter should return a negative answer. However, there is the possibility that a false positive has previously occurred in one of the first $j-1$ filters, in which case the jth packet will be considered by the $j+1$th filter (or discarded if it exhausts all the filters). In this case there will be fewer packets sampled in this flow than desired. In general, for a given flow f, the number of sampled packets is bounded by the minimum between f's packet count and J minus the number of false positives that affects f. Thus a false positive is equivalent to a sampling error and the expected number of false positives $E[F]$ over all the Bloom filters is the appropriate measure to use in evaluating the system:

$$E[F] = \sum_{j=1}^{J} \sum_{i=1}^{n_j} \left(1 - \left(1 - \frac{1}{m_j}\right)^{ki}\right)^{k} \tag{5}$$

where n_j is the *recorded* number of flows with (at least) j packets and m_j is the memory assigned to the jth Bloom filter. It should be noted that n_j will differ from the *actual* number of flows with (at least) j packets due to two factors; the first is that false positives in the $j-1$th filter will effectively be treated as flows of length j (thus increasing n_j) and the second is that false positives may occur in the jth filter (thus reducing n_j). The effect of these factors will be examined in Section V.

In our application, we are clearly interested to find a configuration of the filter parameters that minimizes the number of false positives (i.e., Equation 5). In Section IV we offer a detailed description of an algorithm that computes an estimate of n_j and from that derives the optimal allocation of memory for each filter (m_j). Here we just anticipate one result from that section: the ratio m_j/n_j is constant for any filter j (therefore k is the same across the filters); and we focus on the parameter k which constitutes a complicated factor for accuracy: too many or too few hash functions lead to suboptimal performance.

One approach to minimizing the expected false positives is to differentiate Equation 5 with respect to k and set it equal to 0 to find the global minimum. This method gives us the non-discrete optimal choice of k, but it should be noted that k has to be an integer value. Rounding to the nearest integer is a reasonable fix but does not always result in the best discrete k. However, Dillinger et al. [10] in their work on probabilistic verification of finite-state transition systems, have reported that the value of k that minimizes Equation 5 for a given m_j/n_j (independent of j) can be estimated using a fitted curve:

$$k_{m/n} = \lceil 3.8^{\left(\frac{m}{n}+4.2\right)^{-1}} \frac{m}{n} \ln 2 \rceil \tag{6}$$

Note that we now use this formula to compute k in Equation 5 because we assume the ratio m_j/n_j is constant as will be shown in Section IV.

A. Alternatives to Bloom filters

One alternative to Bloom filter is the Counting Bloom filter [11], where each entry in the filter is not a single bit but rather a small counter. This modification to the standard Bloom filter supports a `remove` operation which allows the number of stored elements to change over time. Even though this would appear ideal in the current application as flows are constantly starting and finishing, it is not appropriate because another storage mechanism would be required to maintain a state for each flow along with its identifier. In fact, such information is needed to remove the flows from the counting Bloom filter after a time out has been reached. This additional mechanism would add complexity, while we desire to design a simple one and easily implementable in hardware. More importantly, it would contrast with the principles behind our scheme where the main idea is to avoid maintaining per-flow information by exploiting the space efficiency of Bloom filters in combination with a time window that excludes the necessity of removing elements from the filters (except at the end of each interval when the filters are emptied).

Bloom filters are not the only probabilistic data structure that can be used to realize the packet sampling scheme. We briefly comment on the use of Multistage filter [12] and k-ary sketch [13] (although an in depth comparison of these methods is beyond the scope of this paper).

A different approach to using a bank of Bloom filters is the Multistage filter (a variation of Counting Bloom filter). A Multistage filter is composed of d hash stages that operate in parallel. A stage is a table of b counters which is indexed by a hash function computed on the flow identifier. Each stage

uses an independent hash function. When a packet comes in, a hash function is computed on its IP tuple for each stage and the indexed counters are incremented by one. If a packet maps to counters of J or more at *all* d stages then this indicates that J packets or more have already been seen for this flow and so this packet is discarded.

In the Multistage filter, there is the possibility that the counters are incremented by more than one flow (i.e., a collision). When a new flow starts, its first packet might map to d counters that are shared with other flows. This condition leads to sample fewer packets for this flow than desired. The main problem for using this data structure in our application is that it is possible that not even a single packet from a certain flow is sampled (i.e., when all d counters are J or more). However, when using a chain of Bloom filters, the event of a false positive in one of the Bloom filter is not likely to cause a false positive on a later filter of the chain because the hash functions are chosen independently at random for each filter; this makes the method more robust to the errors of the underlying data structure. Even though the overall number of collisions in the Multistage filter has been found to be lower [12], the chain of Bloom filters is preferable in this application due to the nature of those false positives.

A k-ary sketch is similar to the Multistage filter. It consists of H hash tables of size m. The hash functions for each hash table are assumed to be chosen independently at random from a class of 2-universal hash functions. The sketch is stored as an $H \times m$ table of registers $T[i][j], i \in [H], j \in [m]$. Denote the hash function for the i^{th} table by h_i. Given a flow identifier x, a k-ary sketch supports the operation `insert(`x`)` which increments the count of bucket $h_i(x)$ by one for each hash table. Let $D = \sum_{j \in [m]} T[0][j]$ be the sum of all updates to the sketch (we arbitrarily use hash table 0 as all hash tables sum to the same value). If an `insert(`x`)` operation is performed for each flow identifier in a packet stream, then for any given flow identifier in a packet stream, for each hash table the value $U_x^i = \frac{T[i][h_i(x)] - D/m}{1 - 1/m}$ constitutes an unbiased estimator for the packet count C_x of the flow identified by x. A sketch can then provide a highly accurate estimate U_x^{est} for any flow identifier x, by taking the median of the H table estimates. In our situation, we would not perform an insert operation for every flow identifier, because we are only interested to count up to J packets for each flow. Therefore, given a flow identifier x, before updating the sketch we could compute U_x^{est} and compare its value with J: if less then the array is updated and the packet is sampled, otherwise the packet is discarded. For the current application, the k-ary sketch presents the same kind of problem as the Multistage filter does: if U_x^{est} is in error at the beginning of a new flow then all the packets belonging to this flow will be discarded.

B. Per-packet level implementation

The abundance of flipflops and ease with which pipelines can be constructed within FPGAs make FPGAs well suited for the implementation of the algorithm we present in this paper. As packet data pass through a series of pipeline stages, the components of the flow identifier can be picked out of the packet at line rate. This is a low cost operation in an FPGA. Once the complete flow identifier has been collected, the $J \times k$ hash functions compute their values in parallel.

It is a feature of some hash functions, such as CRC, that a result is available in as few as 1 cycle and since static memories have similarly low access latencies, in many cases the overall result will be ready even before the whole packet has been received. Computationally, the most expensive component of this process is the calculation of the hash, so system speed (i.e., clock speed) is limited only by the complexity of the hash functions chosen. For example, a 10 Gigabit/s link can be readily accommodated by a pair of 64 bit datapaths clocked at 200 MHz. Such clock frequency is readily feasible in modern FPGA architectures, and there are plenty of hash algorithms that can be made to run at this speed.

Dharmapurikar et al. demonstrated in [14] the feasibility of implementing Bloom filters based packet classification algorithms at OC-192 link rates. The algorithm published there was capable of handling 38 Mpkts/s in hardware using external static RAM. Similarly, the algorithm presented here could make use of external static RAM.

IV. Per-window level memory allocation

This section considers the segmentation of the available memory for allocation to each Bloom filter (i.e., the m_j in Equation 5). There are two issues to be dealt with here; first, Equation 5 is quite cumbersome and is computationally expensive to calculate. Second, the quantities n_j are unknown because they correspond to the number of flows with at least j packets that *will* be seen in the next window, and so these has to be predicted.

The optimum value for the m_j is subject to the constraint that there is a fixed amount of memory available:

$$\sum_{j=1}^{J} m_j = M \tag{7}$$

where M is the total amount of memory available. As far as the authors know a closed form solution for Equation 5 subject to the constraint in Equation 7 is not possible (even using the simplifications known in the literature for the Bloom filter theory). Thus the MATLAB constrained minimization algorithm *fmincon* was applied to find the optimal allocation of the m_j in order to minimize the number of false positives. The optimum m_j are very close to those achieved when $m_j/n_j = m_i/n_i$ with $i \neq j$. This simplification facilitates easy online estimation. Specifically, given that the (near) optimal value of m_j is achieved when $m_j/n_j = m_i/n_i$ with $i \neq j$ implies:

$$m_j - \alpha n_j = 0 \text{ for } j = 1, \ldots, J \tag{8}$$

where α is the required ratio of m_j/n_j. Equations 7 and 8 are linear in m_j and may be easily and quickly solved.

The second problem, predicting n_j is now discussed. There are many time-series forecasting algorithms which can all be applied to this situation (see for example [15]). However, considering the need for online implementation of these

algorithms, we regard the Auto Regressive (AR) model as appropriate for this situation. An AR model consists of linearly regressing on the time-series of interest using the model:

$$\hat{n}_j(k) = \sum_{i=1}^{p} \theta_i n_j(k-i) + \epsilon_j(k) \tag{9}$$

where $\hat{n}_j(k)$ is the predicted value of $n_j(k)$ at time (i.e., window) k, θ_i is the ith parameter associated with ith regressor, $n_j(k-i)$, $\epsilon_j(k)$ is a residual, and p is the order of the model. The order of the model may be estimated using the standard Box Jenkins approach by examining the auto-correlation function and the partial-autocorrelation function of the time-series (see [15] for full details). Given the order of the model, the parameters may then be estimated using ordinary least squares or in the presence of outliers some form of robust least squares (although this would increase the complexity of the implementation).

One complication is that the value of $n_j(k-1)$ is itself only available at the very instant it is required (at the very end of the last window and start of the current window). Thus, no sampling can take place during the time required to compute the estimate for the memory allocation. In order to avoid this situation from arising, either the previous estimate of $n_j(k)$ may used in place:

$$\hat{n}_j(k) = \theta_1 \hat{n}_j(k-1) + \sum_{i=2}^{p} \theta_i n_j(k-i) + \epsilon_j(k) \tag{10}$$

or an estimate based on a partial count of $n_j(k-1)$ may be used (e.g., if we are half way through the previous window we could simply double the count as an unbiased estimate of $n_j(k-1)$).

Finally, consider the value of W. We can assume that n_j depends linearly on W, and so Equation 5 becomes a function of W in $n_j(W)$. Given an acceptable level of false positives (subjectively selected) the appropriate value of W can be easily estimated by use of a search algorithm (as the number of false positives is a function of the window size).

V. Results

Dataset overview

In order to evaluate the sampling scheme, we use traffic traces collected from the edge of a research institute connected to the Internet via a full-duplex 1 Gigabit Ethernet link. In this paper, we present the results obtained by using a 12 hour long trace. In addition, results from other sites where tested and found to give similar results (but are not reported for brevity). We only consider TCP traffic, as it constitutes the majority of the traffic volume for our trace. Figures 4 and 5 show the link utilization and packet rate respectively of TCP traffic measured using a 120 s time interval. The actual maximum link utilization is at around 360 Mbit/s while the maximum packet rate is above 56 Kpkts/s.

Figure 6 plots the (empirical) complementary cumulative distribution function (CCDF) of the number of packets per

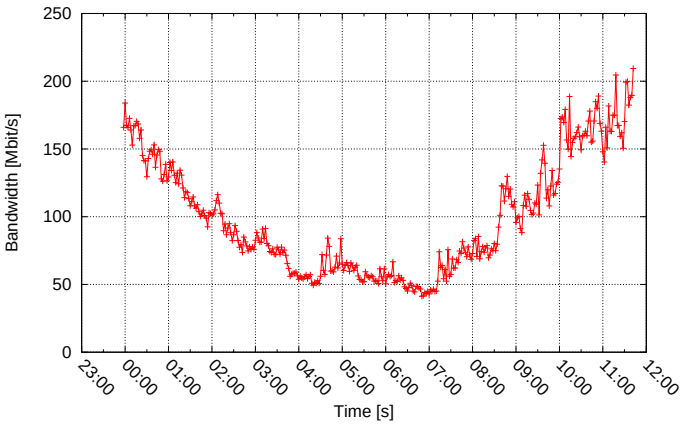


Fig. 4. TCP bandwidth of our reference trace measured using a 120 s time interval.

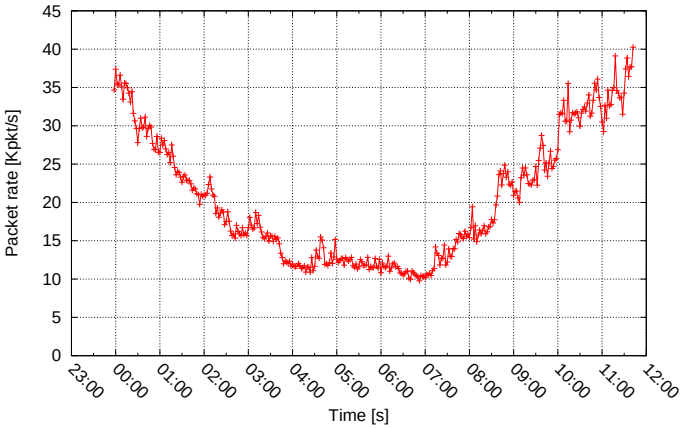


Fig. 5. TCP packet rate of our reference trace measured using a 120 s time interval.

flow for our trace. Note that a "linear" relationship in such log-log scaled plot indicates consistency of the tail with a Pareto distribution. The plot reveals insight about how efficiently sampling up to the J^{th} (with J small) can serve in terms of reducing the volume of data that a flow classification application has to deal with. Further, taking J equal to 10 results in less than 4% of the flows not being sampled completely.

Figure 7 shows the number of active TCP flows measured using a 120 s time interval. The spikes shown in this figure are very likely port and/or address scans.

AR model

We now describe how we obtained the AR model for our trace and present the related results. First, we measured the actual value of $n_1(k)$ for each k, for every 120 s interval of the trace, by substituting an accurate set implementation to the Bloom filters in our scheme. Figure 8 shows the partial auto-correlation function (PACF) for $n_1(k)$ and $n_3(k)$ together

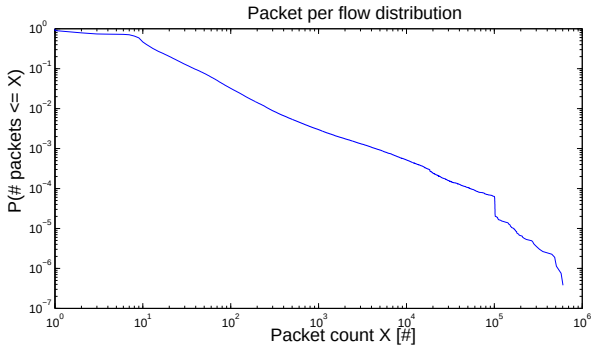


Fig. 6. CCDF of flow packet counts.

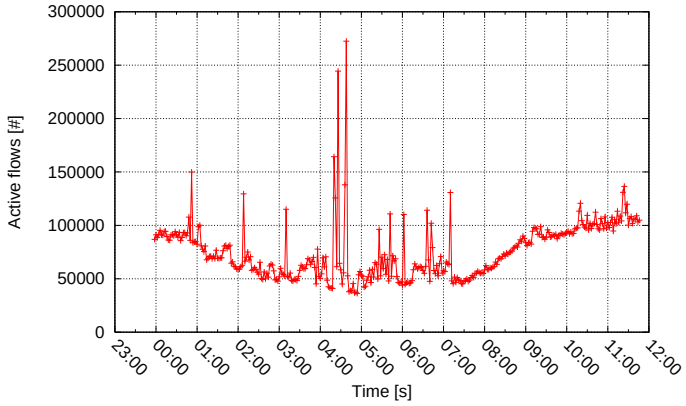


Fig. 7. Number of active TCP flows of our reference trace measured using a 120 s time interval.

with the 95% confidence intervals. As can be seen, there appear to be significant lags at several values for $n_1(k)$. However, once the outliers are removed from the data the PACF resembles that of $n_3(k)$ which shows significant lags only at 1, 2 and 3 implying that an AR model of order 3, AR(3), may be applicable to this time-series. Further, it was found that an AR(3) model appears applicable in all cases (this is not surprising considering the high degree of cross-correlation between the n_j). The forecast and actual values of $n_1(k)$ and $n_3(k)$ are shown in Figure 9 and 10 respectively.

Table I, summarize the parameter estimates and statistics for the AR(3) models trained for 10 Bloom filters. In order to estimate the parameters, ordinary least squares was used and the dataset was split randomly into a training set (2/3 of the data) and a test set (1/3), uniformly distributed across the data. As can be seen there is good agreement between the Mean Squared Errors (MSE) obtained in the training and test sets showing that the models have generalized well. The Mean Absolute Percentage Errors (MAPE) are given for easy interpretation of the results, and these show that on average the forecast is within 4% of the actual.

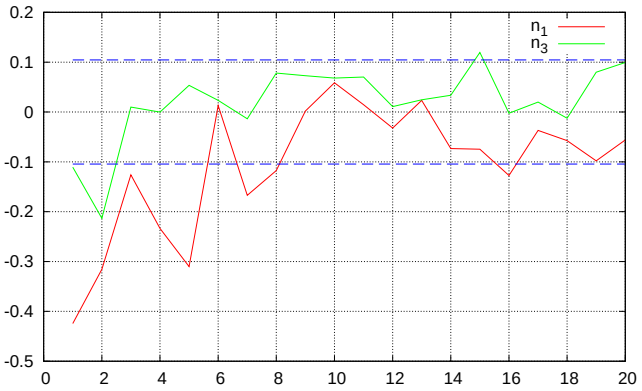


Fig. 8. PACF for n_1 and n_3.

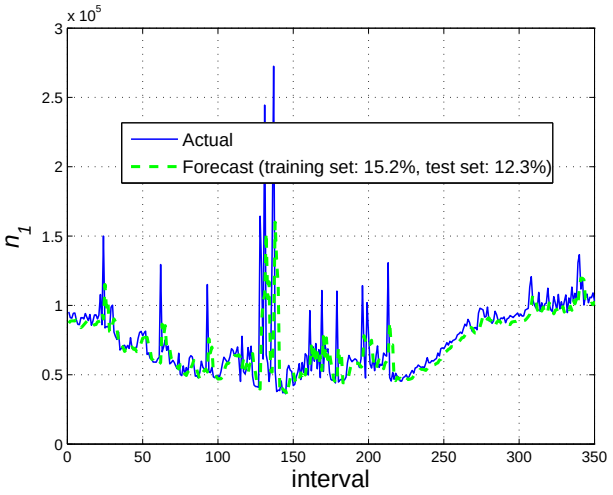


Fig. 9. Forecast and actual values of n_1.

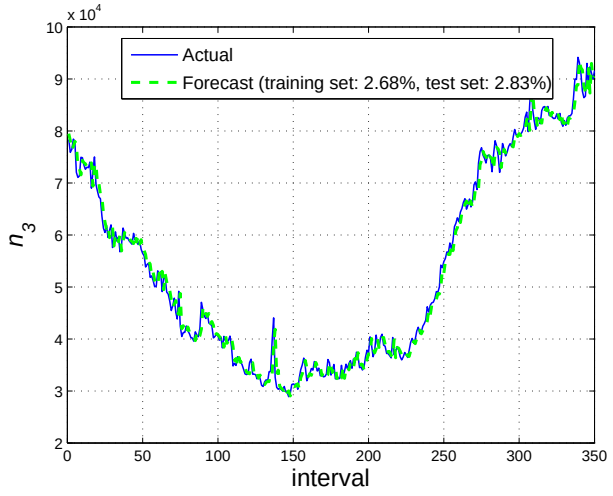


Fig. 10. Forecast and actual values of n_3.

n_i	θ_1	θ_2	θ_3	Training set PMSE	Test set PMSE
n_1	0.65	0.19	0.16	10014	16618
n_2	1.07	-0.17	0.10	3459	4907
n_3	0.79	0.04	0.16	1600	2075
n_4	0.85	-0.07	0.22	1682	1607
n_5	0.79	-0.01	0.23	1601	1759
n_6	0.75	0.01	0.24	1682	1553
n_7	0.81	-0.11	0.30	1442	1580
n_8	0.88	-0.10	0.22	1398	1615
n_9	0.80	-0.13	0.33	1405	1418
n_{10}	0.75	-0.04	0.29	1230	1374

TABLE I
ESTIMATED AR MODEL PARAMETERS AND PREDICTION MEAN SQUARED ERRORS.

Simulations

To evaluate our method we have run several experiments using a software implementation of the packet sampling scheme. By having a software implementation, we are able to add the capability of detecting false positives using hash sets to mirror the set represented by each Bloom filter.

Choosing k independent hash functions for each Bloom filter constitutes a practical problem. In our implementation, we used two different hashing techniques: (i) universal hashing [8], and (ii) enhanced double hashing [10]. Theoretically, universal hashing has the property of behaving as a random function of the input set which ensures uniformity of the output values. The reason for choosing enhanced double hashing over universal hashing is that it only requires the evaluation of two hash functions to generate k independent indices, making the execution faster [16]. Our implementation of enhanced double hashing uses Jenkins' hash function [17] with two randomly chosen initial values to hash a flow identifier into k indices. We found that the results (omitted for brevity) given by using the universal hashing technique are very similar to those obtained with enhanced double hashing, in accordance with [16].

Figure 11 shows the number of false positives, theoretical according to Equation 5 (shown as "expected fp") and measured for two experiments. For these experiments we used J = 10, W = 120 and M = 512KB. Shown as "ratio sol." is the the simulation in which we have used the memory allocation obtained by computing the Equations 7 and 8 for every interval, using the actual n_j previously measured to derive the AR model. This represents the best possible case, i.e., when the forecast of AR model is exactly the actual value. Finally, "ratio ad sol." refers to the simulation of our method using the adaptive memory allocation algorithm. In this case, the estimate of n_j is based on the *recorded* n_j for the past three intervals. As can be seen there is good agreement between the expected number of false positives in all cases. The following figures refer to the "ratio ad sol." experiment.

Unfortunately, the estimate of n_j is not always correct. Figure 12 shows the estimate of n_j over time compared to the recorded value at the end of every interval. These estimation errors yield to sub-optimal memory allocations as it is the case for the traffic spike at about 11:30 in Figure 11. There the measured number of false positives is above the

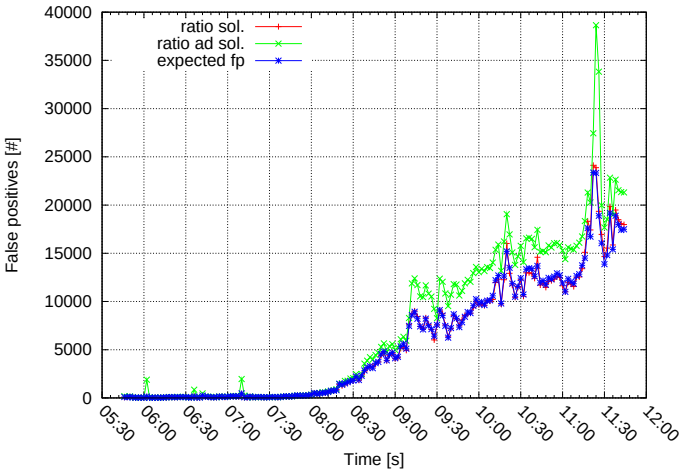


Fig. 11. Number of false positives, measured and theoretical.

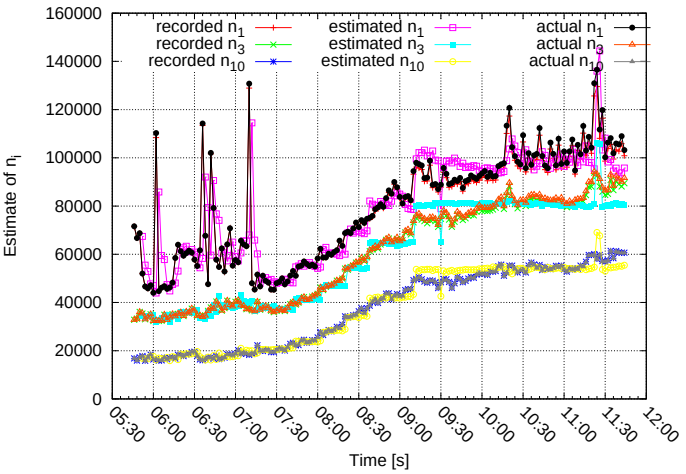


Fig. 12. n_j for $j \in [1, 3, 10]$, estimate and recorded with J=10, W=120 and M=512KB.

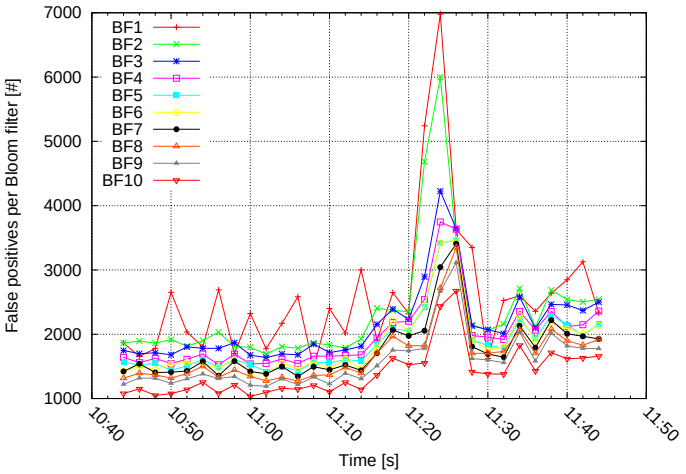


Fig. 13. Number of false positives per Bloom filter, measured with J = 10, W = 120 and M = 512KB.

m_1	m_2	m_3	m_4	m_5
1	0.82	0.72	0.64	0.60
m_6	m_7	m_8	m_9	m_{10}
0.59	0.56	0.53	0.48	0.43

TABLE II
AVERAGE RATIO OF m_j/m_1.

expected one by about 60%. However, when we consider the individual number of false positives per Bloom filter, as shown in Figure 13, it appears that only the first few filters of the chain experience a steep increase of false positives. In contrast, the other filters (which compensate the false positives in early filters by storing more elements), record an evolution of false positives which is in good agreement with the theoretical results.

Figure 14a shows the percentage of sampled packets for the "ratio ad sol." experiment compared to the ideal situation of not having false positives. The analogous plot of the percentage of sampled bytes is represented in Figure 14b. It can be seen that, for both metrics the sampling scheme performance is close to the ideal case, despite the high numbers of false positives shown before. In both figures the "ratio sol." curve is actually below the ideal case. This is because for a small number of flows fewer packets than desired are sampled. Finally, note that this scheme achieves reduction in total traffic in the order of a factor of five in packets, and a factor of ten in bytes.

Table II reports the average ratio of m_j/m_1. These determine the memory allocation and reflect the characteristic of Internet traffic in which the majority of the flows are mice but a small number of large flows account for the quasi majority of the packets.

We now illustrate how the parameter M changes the number of sampled packets and bytes, how many false positives we get and how many flows are affected by false positives and how many packets and bytes which were supposed to be sampled don't get sampled. For the following experiments we only test for 1 hour of the trace, the last hour. Figures 15a and 15b represent the percentage of sampled packets and bytes respectively for M=512KB and multiples. Figure 16 reports the total number of false positives for every interval.

Table III lists the number of flows that are affected by false positives and the amount of packets and bytes that are not sampled. Note that, for our trace, 512KB of memory already provide a good performance, but doubling the memory improves it of an order of magnitude. However, past 1024KB, the benefit given by using more memory quickly decreases.

Testing with a flow classification application

Finally we tested our method with one flow classification application and looked at the results obtained by running with and without sampling. We chose to use the layer-7 traffic classifier (L7) described in [18]. This application uses a pattern matching based technique to classify each flow according to the generating application. L7 uses the information carried in

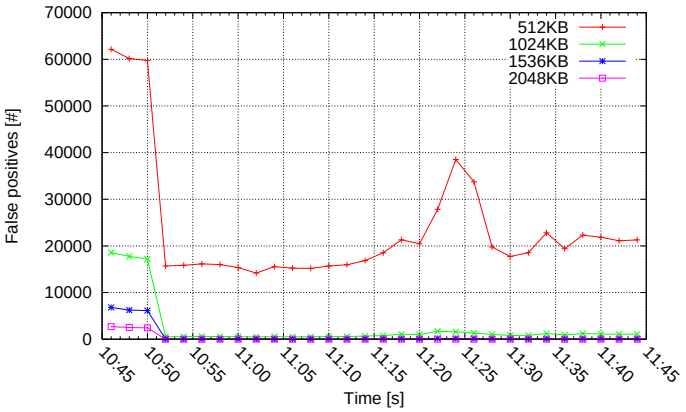


Fig. 16. Number of false positives for M=512, 1024, 1576, 2048KB with J=10 and W=120.

the payload of the few initial packets of each flow. First we run L7 without sampling on the last hour of our trace. It recognizes a total of 2,642,841 flows. Then we feed into L7 the packets sampled by using our method with M=512KB, J=10 and W=120. In this case the total number of flows is 2,636,549. The difference is 6,292 (0.0024%) flows. These are essentially single or two packets flows that are not sampled because of errors due to the false positives in some of the filters. Another 6,021 (0.0023%) flows are instead present among the sampled ones but are not classified by L7, which means that not all their packets could have been sampled, still due to false positives in some of the filters. In total the difference is 12,313 (0.0047%) flows, which causes a negligible loss in the accuracy of L7. However, using this sampling method, L7 experiences a traffic volume which is reduced of an order of magnitude.

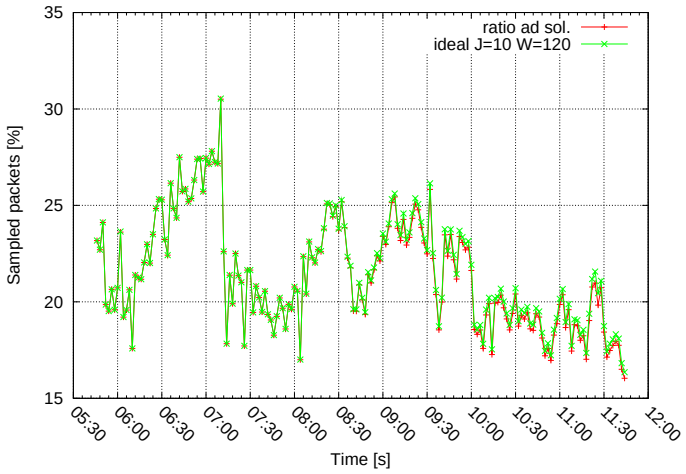
Traffic spikes

Interestingly, the scheme is robust to spikes in the number of active flows caused by attacks or scans (both IP or port scans, as they are regarded as an individual flow each). In fact, during these kinds of attacks there are many more flows having a small packet count, typically 1 to 3 packets. Potentially, this is going to create a high number of false positives in the first three Bloom filters, however, it is likely that these flows will end up in later Bloom filters, assuming J is above 6. Therefore, the scheme is sampling all the first packets of each attack or scan flow with high probability.

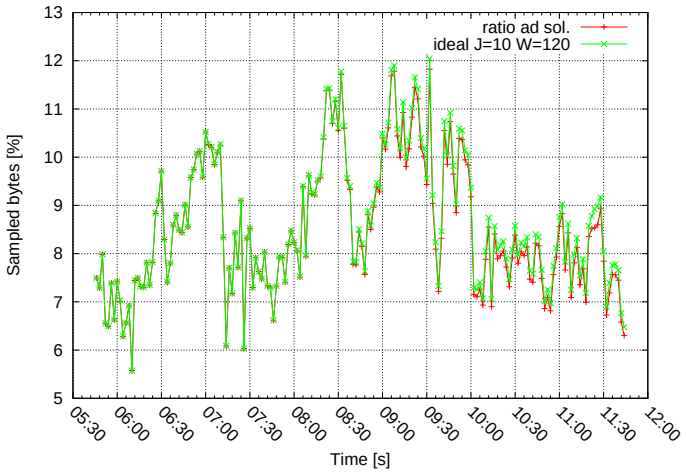
VI. RELATED WORK

Bloom filters have found application in numerous areas of Computer Science, most notably in database applications, and more recently in networking. In [9], Broder et al. have surveyed the network applications of Bloom filters.

Related to our work is the traffic accounting scheme described by Estan and Varghese [12]. They proposed a novel byte-counting algorithm based upon a Multistage filter, a structure derived from Counting Bloom filter [11], which

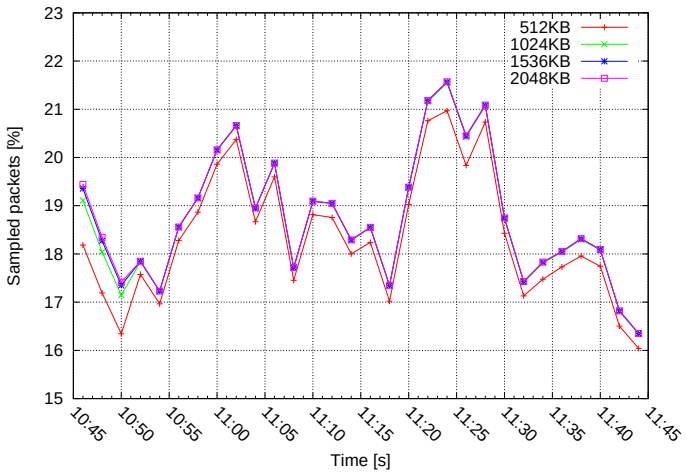


(a) Percentage of sampled packets, ideal versus measured with J=10, W=120 and M=512KB.

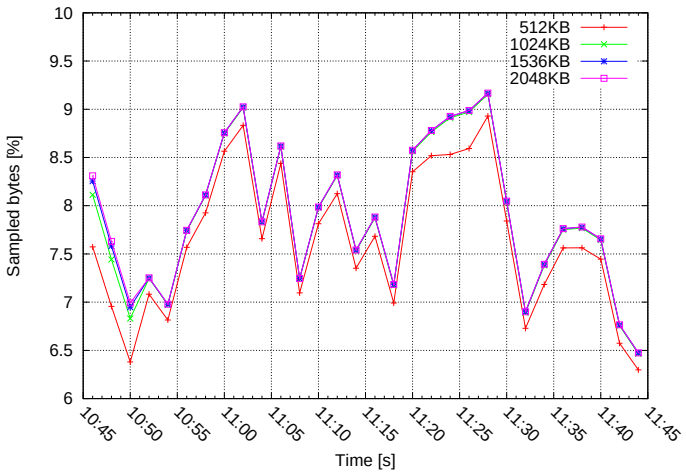


(b) Percentage of sampled bytes, ideal versus measured with J=10, W=120 and M=512KB.

Fig. 14. Sampling metrics for J=10, W=120 and M=512KB.



(a) Percentage of sampled packets for M=512, 1024, 1576, 2048KB with J=10 and W=120.



(b) Percentage of sampled bytes for M=512, 1024, 1576, 2048KB with J=10 and W=120.

Fig. 15. Sampling metrics for J=10, W=120 and M=512, 1024, 1576, 2048KB.

M	Affected flows	Unsampled packets	Unsampled bytes
512KB	5529 (0.0021%)	517173 (0.0221%)	199167936 (0.0323%)
1024KB	4863 (0.0018%)	58007 (0.0024%)	22907748 (0.0037%)
1576KB	4274 (0.0016%)	15830 (0.0006%)	6535918 (0.0010%)
2048KB	4237 (0.0016%)	6086 (0.0003%)	2475934 (0.0004%)

TABLE III
FLOWS, PACKETS AND BYTES THAT ARE AFFECTED BECAUSE OF FALSE POSITIVES.

focuses upon the identification and monitoring of a small number of elephant flows and leads to an implementation optimized for that specific metric. Our work, focusing upon packet-counting, results in a different physical structure.

Kumar et al. [19] introduced a novel data structure, Space-Code Bloom filter (SPBF), which enables approximate per-flow traffic measurements by using multiple Bloom filter with increasing resolutions and extends the Multistage filter approach which addresses the problem of monitoring just a few large flows. In contrast with that work, we use a chain of J standard Bloom filters where J is expected to be a small number (e.g., in the range of 5 to 10). With such few filters it is not efficient to use a multi-resolution SPBF, which is most suited to account for flow sizes.

In [20], the authors propose the Time Machine, a system that uses dynamic packet filtering and buffering to enable bulk recording of large traffic streams. This system implements a filtering scheme that realizes a flow *cutoff*: for every flow, it only keeps up to the first X bytes. Such mechanism is very similar to our sampling scheme, although we use packet count as the unit for flow cutoff. However, our approach is based on probabilistic data structures which can be dimensioned to use just a fraction of the memory occupied by the deterministic data structure implemented in Time Machine.

Finally, an FPGA-based accelerator for Network Intrusion Prevention was presented in [21]. Their approach uses large state tables (maintained in hardware) to define large-volume subsets of traffic which are not interesting for the intrusion prevention system. In our scheme instead, we focus on extracting some packets for each active flow.

VII. Conclusion

We have presented both a method and practical implementation of a flow-sampling scheme suitable for the inline, real-time, classification of high-speed networks. We demonstrated that our approach, based upon the partial sampling of each flow, permits sufficient reduction in total traffic (e.g., a factor of five in packets, a factor of ten in bytes) to permit practical implementations at one Gigabit/s, and, using limited hardware assistance, ten Gigabit/s.

Future Work

The next stage of development for this project is the implementation in hardware and online testing of the system. In addition, the value of W can be adjusted and the advantage of using a bank of bloom filters which overlap the time windows can be investigated. The advantage of this approach would be to determine (to a limited degree) flows which begin in one window and continue into the next.

Acknowledgments

This work was supported by MIUR-PRIN project RECIPE "Robust and Efficient traffic Classification in IP nEtworks". We would like to thank CSITA and GARR who have supported this research.

References

[1] T. Karagiannis, A. Broido, N. Brownlee, kc claffy, and M. Faloutsos, "Is P2P dying or just hiding?" in *IEEE GLOBECOM*, 2004.

[2] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99: Proceedings of the 13th USENIX conference on System administration*, 1999.

[3] W. Li and A. W. Moore, "A Machine Learning Approach for Efficient Traffic Classification," in *Proceedings of the IEEE MASCOTS*, Oct. 2007.

[4] L. Bernaille, R. Teixeira, and K. Salamatian, "Early Application Identification," in *Proc. of ACM CoNEXT*, Dec. 2006.

[5] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *SIGCOMM Computer Communication Review*, January 2007.

[6] N. Williams, S. Zander, and G. Armitage, "A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification," *SIGCOMM Computer Communication Review*, 2006.

[7] N. G. Duffield, "Sampling for Passive Internet Measurement: A Review," *Statistical Science*, vol. 19, no. 3, pp. 472–498, 2004.

[8] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 106–112.

[9] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–609, 2003.

[10] P. C. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[11] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[12] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of ACM Sigcomm*, Aug. 2002.

[13] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of ACM Sigcomm Internet Measurement Conference*, Oct. 2005.

[14] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *ANCS*, Dec. 2006.

[15] J. D. Hamilton, *Time Series Analysis*. Princeton University Press, 1994.

[16] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *To appear in Random Structures and Algorithms*, 2008.

[17] Bob Jenkins's hash function, http://burtleburtle.net/bob/hash/doobs.html.

[18] R. Bolla, M. Canini, R. Rapuzzi, and M. Sciuto, "Characterizing the network behavior of P2P traffic," in *Proceedings of 4th International Telecommunication NEtworking Workshop on QoS in Multiservice IP Networks (IT-NEWS 2008)*, Feb. 2008, pp. 14–19.

[19] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-Code bloom filter for efficient per-flow traffic measurement," in *Proceedings of IEEE Infocom*, 2004.

[20] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a time machine for efficient recording and retrieval of high-volume network traffic," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement (IMC'05)*, 2005.

[21] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: an fpga-based accelerator for network intrusion prevention," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays (FPGA'07)*, 2007.