Building High Accuracy Bloom Filters using Partitioned Hashing

Fang Hao, Murali Kodialam, and T. V. Lakshman
Bell Laboratories
Murray Hill, New Jersey, USA
{fangh,muralik,lakshman}@bell-labs.com

ABSTRACT

The growing importance of operations such as packet-content inspection, packet classification based on non-IP headers, maintaining flow-state, etc. has led to increased interest in the networking applications of Bloom filters. This is because Bloom filters provide a relatively easy method for hardware implementation of set-membership queries. However, the tradeoff is that Bloom filters only provide a probabilistic test and membership queries can result in false positives. Ideally, we would like this false positive probability to be very low. The main contribution of this paper is a method for significantly reducing this false positive probability in comparison to existing schemes. This is done by developing a *partitioned hashing* method which results in a choice of hash functions that set far fewer bits in the Bloom filter bit vector than would be the case otherwise. This lower fill factor of the bit vector translates to a much lower false positive probability. We show experimentally that this improved choice can result in as much as a ten-fold increase in accuracy over standard Bloom filters. We also show that the scheme performs much better than other proposed schemes for improving Bloom filters.

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks; G.3 [**Mathematics of Computing**]: Probability and Statistics

General Terms

Algorithms, Design

Keywords

Bloom filter, hashing

SIGMETRICS'07, June 12–16, 2007, San Diego, California, USA.

1. INTRODUCTION

A Bloom filter is a simple randomized data structure for performing set membership queries. Since a Bloom filter is relatively easy to implement in hardware, there is a growing literature on its use for various networking problems. Examples are in scanning packet-payloads for presence of virus signatures, packet classification using fields beyond the IP header, state look-up and tracking session state for a very large number of sessions [14, 10, 4], routing in peer-to-peer networks, etc. The main reason that Bloom filters are amenable to easy hardware implementation is because the main functions that need to be implemented are hashing, testing and setting bits in a bit vector. These operations have been extensively implemented in hardware for many other applications such as CRC checking. The alternative to using Bloom filters for these applications is the use of associative memories (such as the ternary content addressable memories), hardware implementation of complex pattern matching algorithms, or of other algorithms that are not easy to implement in hardware at high-speeds. The main tradeoff in using a Bloom filter based approach is that the Bloom filter only performs a probabilistic test and that the test can result in false positives. This false positive probability must be tuned to match application needs by appropriate choice of the number of hashes to be performed on each key and the bit vector size (memory used). For high-speed implementation, the available memory is often determined by the amount of on-chip memory in devices such as field-programmable gate arrays that are used to implement the Bloom filter. Ideally, the false positive probability should be made as low as possible given the constraints on the available memory, the number of hashes that can be performed on each key, the number of keys that must be supported. Our goal is to achieve a significant reduction in the false positive probability by a careful choice of hash functions that is well-matched to the input keys.

The motivation for our work is the recent paper of Lumetta and Mitzenmacher [3] which combines the power of two choices with a Bloom Filter to improve the performance of Bloom Filters. The main idea of their paper is for each key to choose one of c sets of hash functions so that the number of ones in the Bloom Filter is reduced. The penalty is that each query has to hash using all c sets of hash functions and the query will be declared to be in the set if it passes any one of these c sets of hash functions.

The false positive rate of the Bloom filter is proportional to the number of ones in the Bloom Filter. Our approach to reducing the number of ones is to search for a good set of hash functions for a regular Bloom Filter. A simple brute force approach to search for good hash functions will not work due to the concentration bounds on the number of ones in the Bloom filter. This implies that we have to search an enormous number of hash functions to get a set of hash functions to find something that results in a small number of ones. The way we achieve this is by partitioning the keys into subsets and using a different set of hash functions for each subset. This leads to an exponential increase in the set of available hash functions and searching for a good hash function in this expanded space is much easier. In fact, we show that even a greedy algorithm for choosing appropriate hash functions results in a Bloom filter with excellent performance. Unlike the approach in [3] where each query has to hash using c sets of hash functions, in our case each query has incurs only one additional hash compared to a regular Bloom filter.

A summary of our contributions is as follows:

- We develop a *Partitioned Hashing* approach, in which the set of input keys is partitioned into disjoint subsets and a different set of hash functions is used for each subset. The partitioning of the set into disjoint subsets is done using an additional hash function.
- We show that this mechanism results in significantly better performance, sometimes by an order of magnitude, than standard Bloom filters.
- We have first shown the feasibility of our approach using perfect random hashing. We have also implemented our scheme using a simple CRC based hashing mechanism which matches the performance of random hashing both on synthetic as well as trace data.

2. MOTIVATION

A Bloom filter is a randomized, memory efficient data structure for performing membership queries. Let S be a set comprising of n keys. The Bloom filter stores this set in a bitmap (filter) with m bits by hashing each element in S into the bitmap using k independent uniform hash functions $h_1, h_2 \ldots h_k$. A bit in the filter is set to one if and only if one or more keys hash to that location in the bitmap. If we want to check whether some given x belongs to the set, we compute $h_1(x), h_2(x), \ldots h_k(x)$ and declare $x \in S$ if all these k bits are set to one in the filter. Assuming uniform hash functions it easy to determine the probability that a query membership test will result in a false positive. In this paper, we consider three different notions of false positive probability.

DEFINITION 1. *Given the number of keys n, the number of bits in the bitmap m and the number of hashes k, the a priori false positive estimate or the false positive estimate before hashing p_b, is the expected false positive probability for the given set of parameters.*

Assuming that the hash functions are uniform, it is easy to show that

$$p_b = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k.$$

Setting the partial derivative of p_b with respect to k to zero, gives the number of hash functions that minimizes the false positive probability. This is attained when

$$k = \log_e 2 \, \frac{m}{n}$$

and this number is rounded to an integer to determine the optimal number of hashes. Note that the definition of p_b does not involve knowing exactly how many bits are set to one in the Bloom Filter. This is due to the fact that the number of ones in the Bloom Filter is strongly concentrated around its mean and the formula for p_b accurately predicts the expected performance of the Bloom Filter. In order to more accurately model the performance of the Bloom Filter we define the *fill factor* of the Bloom filter to be the factor of bits that are set to one after all the keys are hashed. The actual performance of the Bloom filter is a function of the fill factor.

DEFINITION 2. *The posterior false positive estimate or the false positive estimate after hashing p_a is the expected false positive probability if the fill factor of the Bloom filter is known.*

If f is the fill factor of the Bloom filter, the posterior false positive estimate of the Bloom Filter is

$$p_a = f^k.$$

The posterior false positive estimate is a much better estimator of the actual performance of the Bloom filter. In practice, due to the concentration around the mean, the prior and the posterior estimates are usually very close. In this paper, we try to find hash functions that give lower fill factor and hence give better performance that what is given by p_b.

DEFINITION 3. *The observed false positive probability p_o, is the actual false positive rate that is observed when queries are made on the Bloom filter.*

In our case the observed false positive value is computed via simulation. Note that the observed false positive value is an experimental quantity and not a theoretical estimate. Our objective is to minimize the observed false positive probability. Since we expect the observed false positive probability to be close to the posterior false positive value, we choose hash functions to minimize the fill factor. Towards this end, assume that we have a large (potentially infinite) number of uniform hash functions $h_1, h_2, \ldots$, which takes any key and maps it uniformly onto the filter with m bits. The objective, now is to pick k hash functions such that the Bloom filter has a small fill factor. We first show that a brute force search is not likely to be a good approach to find a set of hash functions that reduces the fill factor. This is due to the fact that the number of ones in the filter is

strongly concentrated around the mean. Note that when we hash kn times into a filter of size m, the expected fill factor is given by $\left(1 - e^{-\frac{kn}{m}}\right)$. The next result shows [5] that the number of ones is concentrated tightly around the mean.

THEOREM 1. *Let Y represent the number of bits that are set to one when n keys are hashed with k hash functions into a filter with m bits. Then $E[Y] = m\left(1 - e^{-\frac{kn}{m}}\right)$ and*

$$Pr\ [|Y - E[Y]| > \lambda] \leq 2e^{-\frac{\lambda^2}{2kn}}.$$

Actually the situation is even worse than what is indicated by the above result since this bound is not tight, especially close to the mean. In order to illustrate this, we consider a problem with $n = 1000$ keys that have to be hashed onto $m = 8000$ bits. The optimal value of $k = 6$. Therefore we hash 6000 times into a filter with 8000 bits. The expected number of bits that will be set equal to one is 4221.06. We repeated this experiment 10^7 times. Figure 1 shows the left tail of the probability distribution function along with the bound shown in Theorem 1. Note that the actual probability density function falls of several orders of magnitude faster than the exponential tail given in Theorem 1. This makes it extremely difficult to do a brute force search for hash functions with low fill factor. This effect is also highlighted in Figure 2 where we show the minimum fill factor as a function of the number of experiments. The mean number of ones is 4221.06 with a corresponding fill factor of 0.528. The apriori false positive estimate is $p_b = 0.0216$. After searching through 10 million uniform hash function sets, the minimum number of ones in the filter is 4089 (fill factor of 0.511). This corresponds to a false positive probability of 0.0178. Finding lower values than this gets successively more difficult since the rate of decrease of the best fill factor slows down exponentially. Instead, we increase the state space by partitioning the keys and using different sets of hash functions for each key. We illustrate this approach and its benefits by analyzing a simple case.

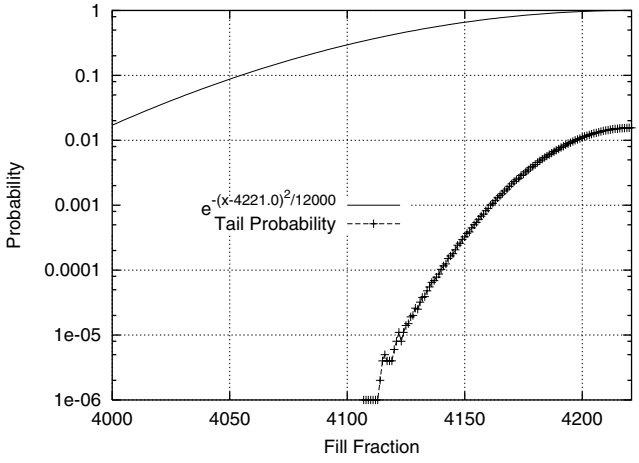

Figure 1: Comparison of the Tail Probability and Concentration Bound: $n = 1000$, $m = 8000$, $k = 6$

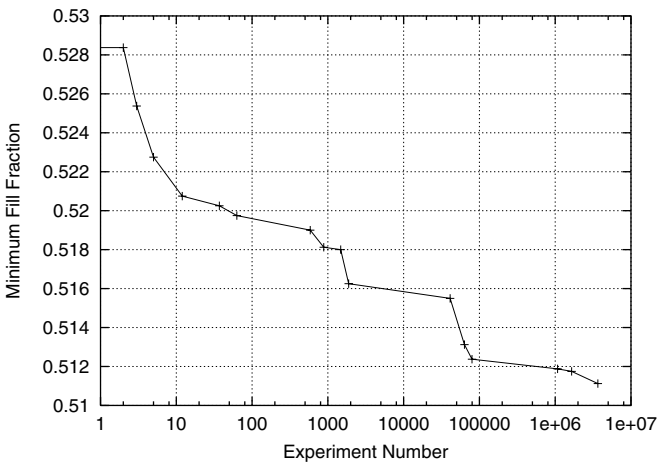

Figure 2: Minimum Fill Factor versus Number of Experiments

3. BALLS AND BINS ANALOGY

A frequently used analogy for hashing is randomly placing balls in bins. The balls represent the keys and the bins represent the hash. Any random placement of balls can be viewed as a uniform hash function. Assume that we have n balls that are placed into m bins. Our objective is to minimize the number of bins that have at least one ball since this represents the fill factor of the hash. We consider two different approaches. In the first approach, all n balls are placed at random into the m bins and the fill factor is observed. The experiment is repeated a number of times and the experiment that minimizes the fill factor is chosen. The mapping corresponding to this minimum fill factor is the hash that gives the best fill factor. This corresponds to a brute force approach to picking a good hash function. We now outline an alternate approach. The connection of this approach to the hashing problem will be made clearer in the next section, but for now let us assume that we are interested is minimizing the fill factor. In this approach the first ball is tossed into a bin selected at random. For each subsequent ball, we choose a bin at random among the m bins and if it already has a ball, then we place our ball into that bin. If the bin is empty then we pick another bin. This process of finding a bin with a ball in it or finding another bin at random is repeated at most t times. The process ends when we either find a bin with a ball in it and we place the current ball into the bin, or all t bins chosen are empty in which case the ball is placed in the last chosen bin. (Actually, we can place the ball in any of the t bins that were picked). The balls in bins analogy has been used in load balancing but there the objective is to distribute the balls evenly in the bins. In our case we want the opposite. Our objective is to pack all the balls into as few bins as possible. Let Y_j represent the number of non-empty bins after j balls have been dropped. The value of $Y_1 = 1$ and

$$Y_j = \begin{cases} Y_{j-1} & \text{with probability } 1 - \left(1 - \frac{Y_{j-1}}{m}\right)^t \\ Y_{j-1} + 1 & \text{with probability } \left(1 - \frac{Y_{j-1}}{m}\right)^t \end{cases}$$

We are interested in determining the expected fill factor $\frac{1}{m}E[Y_n]$. It is easy to see that

$$\frac{1}{m}E[Y_j] = \frac{1}{m}E[Y_{j-1}] + \frac{1}{m}E\left[\left(1 - \frac{Y_{j-1}}{m}\right)^t\right]$$

If we consider the case where $n << m$, and replace the non-linear term with the first term in the Taylor series expansion,

$$\left(1 - \frac{Y_{j-1}}{m}\right)^t \approx 1 - t\frac{Y_{j-1}}{m}$$

and letting $f_j = \frac{1}{m}E[Y_j]$, we get

$$f_j \approx f_{j-1} + \frac{1}{m}\left(1 - tf_{j-1}\right)$$

This is a linear difference equation that can be solved to give

$$f_n \approx \frac{1}{t}\left[1 - \left(1 - \frac{t}{m}\right)^n\right].$$

We plot the above equation for $n = 3000$ and $m = 8000$ for different values of t. Note that the expected minimum fill factor falls off with t. The non-linear effects start to matter for large values of t and the rate of decrease levels off. Also note that the minimum fill factor corresponding to $t = 1$ the expected fill factor when balls are added at random. In order to put this decrease in perspective, we ran 10^6 experiments with the same parameters. The expected fill factor is 0.3127 and the minimum fill factor after a million experiments is 0.301. Note that even for $t = 2$ the expected fill rate is 0.26. We could not get the fill rate down to this value with $t = 1$ even after 10^9 experiments. The reason for this huge performance improvement is the following: When we make t attempts to throw each ball and there are n balls in all, we are actually exploring t^n different sample paths of the ball tossing experiment. The paths are not independent. Further, we just use a greedy algorithm to find a sample path with a small fill factor. However, the fact that the number of sample paths grows exponentially with t gives a low fill factor for the algorithm. In the next two sections, we extend this idea to a practically implementable algorithm to reduce the posterior false positive probability of Bloom filters.

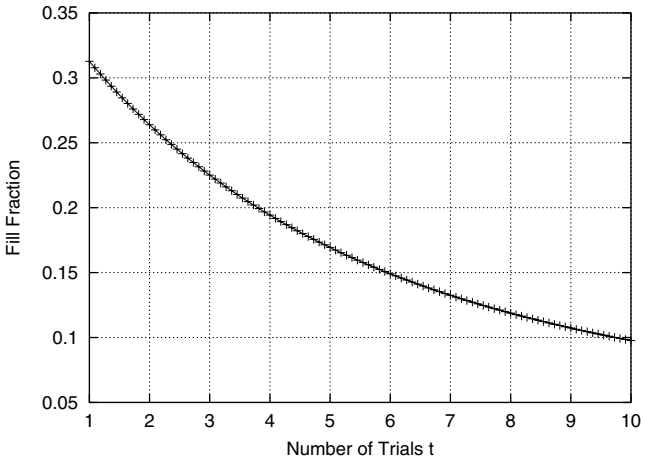

Figure 3: Expected Fill Factor as a Function of the Number of Trials t : $n = 3000$, $m = 8000$

4. PARTITIONED HASHING

The balls in bins problem considered in the last section, can be viewed as a hashing scheme where each key uses a different hash function. There are two problems with this idea.

- The first is that when a membership query is made it is not clear which set of hash functions to use. In [3], each key uses one of r hash functions and each query uses all these hash functions and returns a positive if any of these r tests results in a positive result. In our case, we partition the keys into disjoint subsets and use a different set of hash functions for each set. The partitioning of the keys into different subsets is done via a hash function. When a membership query is made, it is first hashed to the determine which set of hash functions will be used by this query. Then the hash functions corresponding to this group will be used by the query.
- The second problem with the simple example outlined in the last section is that specifying all the hash functions also takes memory that has to come from the m bits that we have for the filter.

Figure 4 gives a picture of partitioned hashing where the set S is partitioned into 5 partitions (groups). Note that the hash functions for each group are independent. *The hash functions for different groups can be dependent.* Note that h_1 is used by both groups 1 and 4. Also note that the number of hash functions can be different for different groups. Indeed this flexibility can also be exploited to minimize the overall false positive probability. However, in the rest of the paper, we assume that the number of hash functions is the same for all groups and is fixed at the optimal k for the Bloom filter parameters.

To understand the effect of the number of groups, as well as the number of hash functions that we need to search for each group to pick the k best hash functions, we ignore the problem of the overhead for the hash functions and focus on the problem where the n keys are partitioned into g equal groups each with $b = n/g$ keys. We assume that g is chosen such that b is integral. As in the case of the ball in bins problem we analyze the minimum fill factor achieved by a natural greedy algorithm. Unlike the balls in bin problem where each ball chose the best of t bins, in the partitioned hashing case each group has to choose k hash functions. Therefore each trial for a particular group is to pick k hash functions and check the fill rates given by this set of hash functions. This experiment is repeated t times and the hash

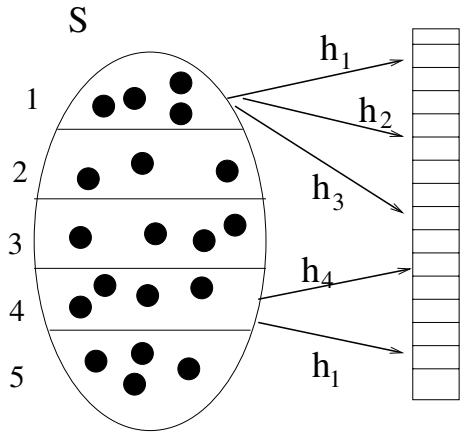

Figure 4: Partitioned Hashing

set the minimizes the fill factor is chosen. We now describe this process in some more detail: All the bits in the Bloom filter are initialized to zero. We hash the first group t times and pick the one that minimizes the fill factor. Note that each experiment consists of hashing each key in the first group k times. The set of hash functions for the first group is now fixed and the corresponding bits are set in the Bloom filter. The same process is repeated for the next group. When we process group j, all the keys from groups 1 to $j-1$ are hashed onto the Bloom filter. We do the following for group j.

- Each key in partition j is temporarily hashed k times onto the Bloom filter and the fill factor of the Bloom filter is observed. All the keys in partition j are unhashed.
- The experiment in the last step is repeated t times.
- The keys in group j are permanently hashed onto the Bloom Filter using the experiment that results in the smallest fill factor.

This process is repeated until all the groups are hashed. For each group, the best hash set is picked out of t trials. The performance of the greedy algorithm will depend on the value of t and we would expect the performance of the algorithm to improve with t. The fill factor output by the algorithm is a random variable and we use the expected value of the fill factor to measure the performance of this greedy algorithm. Let Y_j represent the number of ones in the Bloom filter after j groups have been hashed. When group $j+1$ is hashed, there will be inter-group overlap with the ones from the previous j groups and intra-group overlap between the hashes belonging to keys in group $j+1$. In order to simplify the analysis, we ignore the intra-group overlap. This will lead to an overestimate of the fill factor. If the number of groups g is not too small, then this simplification does not lead to a significant loss of accuracy. Comparing the results of our analysis with simulation results, we show that the results that we obtain are quite accurate. Let N_j represent the number of new ones introduced into the Bloom filter by group j. The evolution of the number of ones in the Bloom filter follows the equations:

$$\begin{aligned} Y_1 &= b \\ Y_j &= Y_{j-1} + N_j \quad j = 2, 3, \ldots, g \end{aligned}$$

Note that $Y_1 = b$ since we are ignoring intra-group overlap. Let

$$f_j = \frac{1}{m} E[Y_j].$$

Therefore

$$\begin{aligned} f_1 &= \frac{b}{m} \\ f_j &= f_{j-1} + \frac{1}{m} E[N_j] \quad j = 2, 3, \ldots, g \end{aligned}$$

We write $E[N_j] = E[E[N_j|Y_{j-1}]]$ and now determine the value of $E[N_j]$ by setting $Y_{j-1} = \alpha$. Let Z_{jl} represent the number of ones when we run experiment l for group j.

$$\Pr[Z_{jl} = w] = \binom{bk}{w} \left(\frac{\alpha}{m}\right)^{bk-w} \left(1 - \frac{\alpha}{m}\right)^{w}.$$

Since the right hand side is independent of l we represent the right hand side as δ_w. Observe that $N_j = 0$ if $Z_{jl} = 0$ for at least one experiment l and this happens with probability $1 - (1-\delta_0)^t$. In general, $\{N_j > p\}$ if and only if $\{Z_{jl} > p\}$ for all l. Therefore $\Pr[N_j > p] = (1 - \sum_{w=0}^{p} \delta_w)^t$ for $p = 0, 1, \ldots, bk$. This implies that

$$\begin{aligned} E[N_j|Y_{j-1} = \alpha] &= \sum_{p=0}^{bk} \Pr\left[N_j > p | Y_{j-1} = \alpha\right] \\ &= \sum_{p=0}^{bk} (1 - \sum_{w=0}^{p} \delta_w)^t \end{aligned}$$

We now have to uncondition Y_{j-1} and this is the expectation of a non-linear term. We approximate this term by passing the expectation into the non-linear expression. We define

$$\beta_{jw} = \binom{bk}{w} f_j^{bk-w} \left(1 - f_j\right)^{w}.$$

Putting everything together, the performance of the greedy algorithm can be determined by solving the following equations iteratively:

$$\begin{aligned} f_1 &= \frac{b}{m} \\ f_j &= f_{j-1} + \frac{1}{m} \sum_{p=0}^{bk} (1 - \sum_{w=0}^{p} \beta_{jw})^t j = 2, 3, \ldots, g \end{aligned}$$

The approximation works extremely well over the entire range of parameter value that we tested. In Figure 5, we plot the evolution of f_j obtained from simulation and compare it with the theoretical value. Note that even for a small number of groups the prediction is very accurate.

We now study the effect of group size g and the number of trials t on the posterior false positive probability. Figure 6 shows the false positive probability as a function of the number of groups. Each curve represents a different value

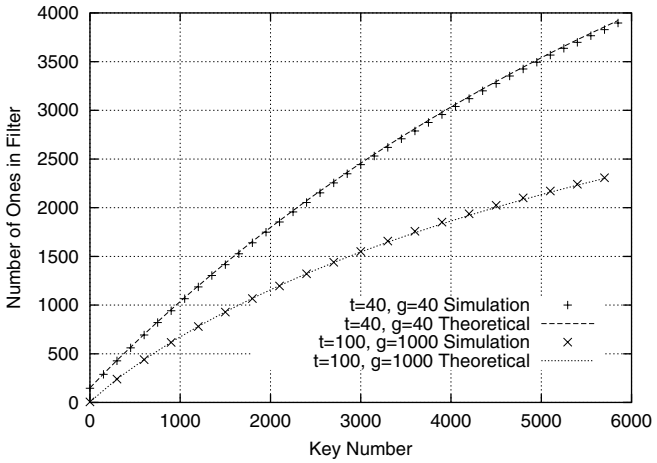

Figure 5: Comparison of the Theoretical and Experimental Performance: $n = 1000$, $m = 8000$, $k = 6$

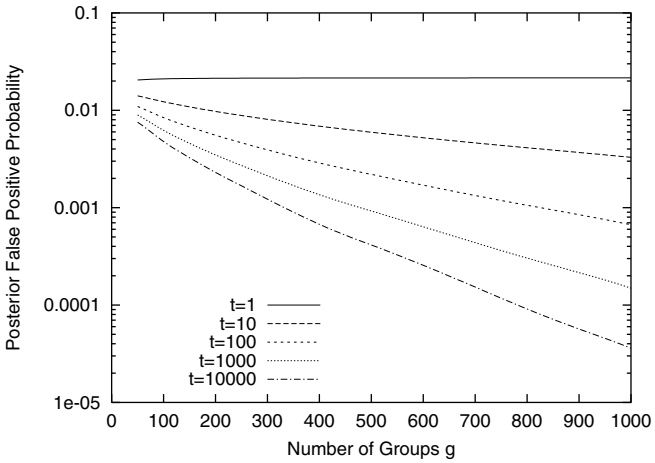

Figure 6: Fill Factor as a function of number of groups g**:** $n = 1000$, $m = 8000$, $k = 6$

of t. Note that the false positive probability drops rapidly (the y-axis is on a log scale) as the number of groups increases, for all values of t except $t = 1$. (Note that $t = 1$ is the case where the hash function is picked at random.) The rate of decrease of the false positive probability is greater for larger values of t. Figure 7 shows the variation of the false positive probability with the number of trials. Each curve represents a different group size. Here the decrease is more gradual for increasing values of t and the decrease is more rapid for larger values of g. These plots are consistent with the observation that the number of sample paths considered is roughly t^g. This leads to the exponential increase in the number of combinations with increasing g and consequently an exponential decrease in the false positive rate with increasing g. The effect of changes in t on the false positive rate for a fixed g is polynomial and the rate of decrease increases with increasing g. The results are suggestive of the t^g relationship but it needs more study to determine the relationship between the number of sample paths and the false positive probability.

There are three practical issues that we have to deal with in order to make the scheme practical:

- In all the analysis above, we assumed that we have an potentially infinite set of hash functions. In reality we have a fixed (perhaps large) number of hash functions that we use and each group picks hash functions from this set. We represent the total number of hash functions available by H out of which each group chooses k hash functions. Therefore there are potentially $\binom{H}{k}$ combinations of hash functions that any group can use.

- Since each group potentially uses different hash functions, we have to store pointers to the hash functions. Therefore each group uses $k \log H$ bits to indicate the k hash functions that it is going to use.

- Though the analysis indicated that having a large number of groups is desirable, since each group uses $k \log H$ bits of memory this has to be taken into consideration when the group size is chosen. Indeed, this penalty has a significant effect and therefore the group size that minimizes the false positive probability is usually not too large.

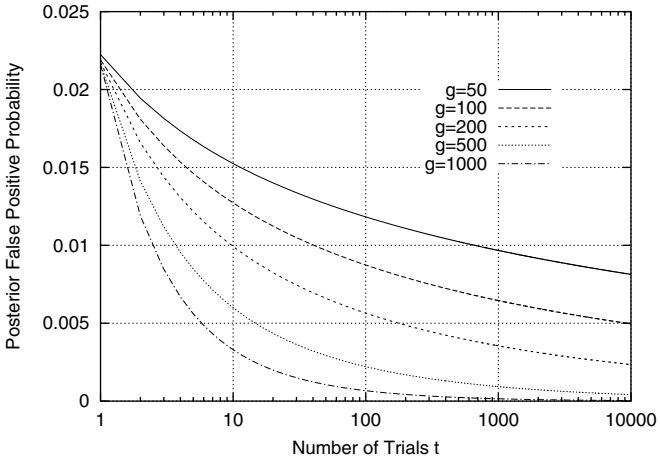

Figure 7: Fill Factor as a function of number of trials t**:** $n = 1000$, $m = 8000$, $k = 6$

We also observe that:

- The bit penalty for increasing H is only logarithmic. Therefore in practice, it is better to have a moderate value of g and increase H instead.

- There is no reason to just use a one pass greedy algorithm for finding a good hash function. If we are given H hash functions and each group has to pick k of these hash functions, the objective is to minimize the fill factor. We can formulate an integer programming problem to solve this problem as follows: Let

$X_{ij} = 1$ if group i picks hash function j and $X_{ij} = 0$ otherwise. Let $Z_l = 1$ if bit l is set to one and zero otherwise. We say $l \in (i, j)$ if bit l is set to one if group i picks hash j.

$$\min \sum_{l=1}^{m} Z_l$$

$$\begin{aligned} \sum_{j=1}^{H} X_{ij} &= k \quad \forall i \\ Z_l &\geq X_{ij} \quad \text{if } l \in (i,j) \\ X_{ij} &\in \{0,1\} \end{aligned}$$

This is a generalized version of set cover and is therefore NP-complete. Further from the experiments, as the number of available hash functions H increases, the integrality gap increases. Therefore solving the LP relaxation is not very useful. Therefore we use a multi-pass heuristic to solve this problem.

The implementation details including the hash functions as well as the heuristic to minimize the fill rate is described in the next section.

5. IMPLEMENTATION DETAILS

In this section, we describe implementation details of the Partitioned Hashing mechanism. In order to show the feasibility of this approach in practice, we also provide a simple yet effective implementation by using a set of CRC based hash functions.

5.1 Details of Partitioned Hashing

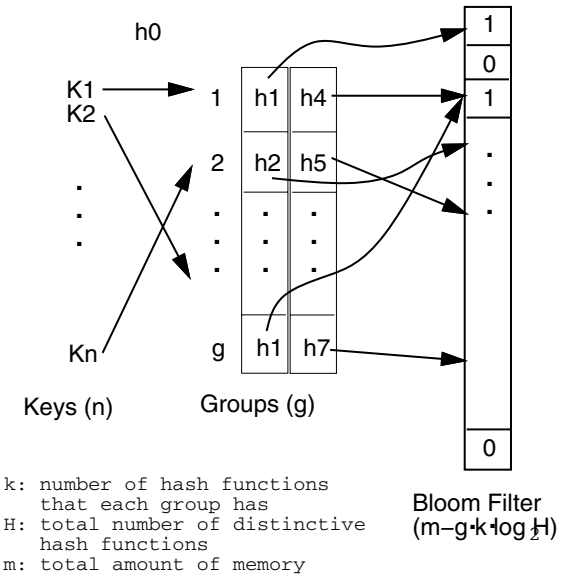

Figure 8: Partitioned Hashing

Partitioned Hashing is done in two steps, as illustrated in Figure 8. In the first step, each key is hashed into one of g groups by using hash function h_0. Each group is associated with k independent hash functions. Note that the same hash function can be used by different groups. For example, as shown in Figure 8, both group 1 and group g use h_1. In the second step, each key is hashed k times to the Bloom filter by using the k hash functions of its group, and the corresponding bits in the filter are set to be one.

After the filter is generated, the new arrivals are processed following a similar procedure. They are first hashed into a group by using h_0, and then hashed k times into the Bloom filter by using the k hash functions of the group. The arrival is considered in the set if all k corresponding bits in the filter are ones; and not otherwise.

This approach needs $g \cdot k \cdot log_2 H$ extra bits to identify the selected hash functions of each group. Recall H is the total number of hash functions in the system. Given that the total amount of memory is m bits, only $m - g \cdot k \cdot log_2 H$ bits are left for the Bloom filter on the second stage. This is the cost that we pay for the gain of flexibility in allowing different groups to use different sets of hash functions. We will show in the experiments that the reduction of fill factor resulted from better choice of hash functions for each group more than compensates the cost in memory.

We use a simple greedy algorithm to select k out of H hash functions for each group, with the goal of minimizing fill factor of the filter. As shown in Figure 9, the algorithm goes through a few iterations until fill factor can no longer be reduced. During the initialization phase, we pick the best hash functions one by one for each group, selecting the function that gives the lowest fill factor. This phase ends after all groups are processed. (This step was also described previously in Section 4.) We can then further improve the results by going through more iterations, in which we remove the hash functions for each group, and then re-select each function one by one. We use a counting Bloom filter to implement the insertion and deletion of keys as follows:

When a key is hashed to a Bloom filter bit, we increase the corresponding counter. When a key is deleted (i.e., unhashed) from a bit, we decrease the corresponding count. Any positive count means the bit is set; count of zero means otherwise. Note that the counting Bloom filter is just used for the purpose of searching hash functions offline. It will no longer be used once the Bloom filter is formed based the existing set of keys and selected hash functions.

One implementation issue is whether it is possible to find a large group of independent hash functions that are simple, and preferably easy to implement in hardware. In the next section we use one example to show this is indeed feasible.

5.2 CRC-based implementation

Hashing and Bloom filters have been widely applied to the domain of data networking and computer security [4]. For such applications, the keys are usually strings of various lengths. For instance, in IP packet content inspection problem, one may want to test if an arriving packet contains a suspicious payload.

To address such applications, we define both our first stage and second stage hash functions as mappings from strings

INITIALIZATION

1. For eachgroup i, where $i = 1 \dots g$
1.1 For each h_j, where $j = 1 \dots H$
1.1.1 Hash all keys of the group into Bloom filter
1.1.2 Count the fill factor
1.1.3 Remove all keys of the group from filter
1.2 Choose the hash function h_{j_1} with minimum fill factor
1.3 Use h_{j_1} to hash all keys of the group into Bloom filter
1.4 Repeat Steps 1.1 through 1.3 k times to select k different functions for group i

ITERATIONS

2. For each group i, where $i = 1 \dots g$
2.1 For each h_{j_l}, where $l = 1 \dots k$
2.1.1 Remove all keys of the group from filter
2.2 Repeat Steps 1.1 through 1.4 to re-select k different functions for group i

3. Repeat Step 2 until there is no more change in fill factor

Figure 9: Description of the greedy algorithm for hash function selection

to integers. The first stage function $h_0(key)$ maps a key to a group:

$$RS(key) \ mod \ g$$

where RS is the hash function from Robert Sedgewicks's book [6][1].

The second stage function $h_j(key)$ maps a key to a bit in Bloom filter:

$$CRC(CRC(0, key),\ string(R_1 + j \cdot R_2)) \ mod \ m$$

where $j = 1 \dots H$. R_1 and R_2 are two constant random numbers [2]. Function $CRC(crc, s)$ takes the four-byte integer crc as seed, and derives the 32-bit CRC checksum of the string s in addition to the seed. Function $string(i)$ converts integer i to four-byte string. Observe that $CRC(0, s)$ generates the CRC checksum of string s, and $CRC(CRC(0, s_1),\ s_2)$ generates the CRC checksum of the concatenated strings s_1 and s_2.

There are several nice features of this design:

- Both RS and CRC are of complexity $O(l)$, where l is the length of the key.
- For each key, we just need to compute $CRC(0, key)$ once. All second stage hash functions $h_j(key)$ can be derived by using $CRC(0, key)$ as the seed. Since $string(R_1 + j \cdot R_2)$ just has four bytes, the additional calculation for each hash function $h_j(key)$ is minimal.
- The only data to store for computing all the h_j's are two random numbers R_1 and R_2. So there is essentially no hardware limit on the total number of hash functions H that the filter can use.
- We will show in experiments that this set of hash functions perform almost the same as independent uniform random hashing when used in Bloom filters. Kirsch and Mitzenmacher have given the theoretical analysis of linear combination of hash functions in [2].

Note that it is important to use different types of hash functions for first and second stage to avoid unexpected correlations.

5.3 Further discussion

This mechanism naturally fits in applications where change in set of keys occurs at a much longer time scale than membership tests for new arrivals. For example, for firewalls that perform content based filtering, the set of keys may change on the order of weeks or days, while membership testing has to be done at line speed.

In such applications, when keys are added to the set, we can simply set the corresponding bits by used the existing hash functions. When keys are deleted, we can simulate the deletion offline by using counting Bloom filter, and then modify the actual Bloom filter bits correspondingly. Optimization can be done offline either periodically, or after a certain number of additions and deletions occur. The result of optimization can then be used to reset the Bloom filter.

6. EXPERIMENTAL RESULTS

We evaluate our approach in the following three steps. First, we study the impact of parameters such as number of groups g and total number of hash functions H on the false positive probability. Second, we compare the Partitioned Hashing (PH) approach with both standard (STD) Bloom filter and Power of Two Choices (PTC) Bloom filter. We show that with the same amount of total memory cost, PH has significant improvement over both prior schemes. We use uniform random hash along with randomly generated synthetic trace in both the above two steps. In the last step, use the CRC based hash functions, and apply the PH approach to a real packet trace that contains full payload. We show that our approach is very easy to implement in practice, and the CRC based hash functions generate the same false positive probability as the uniform random hashing.

6.1 Filter configuration

Both number of groups g and total number of hash functions H have a significant impact on the false positive probability. Larger g and H lead to more possible combinations of different hash functions, and hence better chance of reducing the filter fill factor. On the other hand, larger g

[1]The functions is also available at http://partow.net/programming/hashfunctions

[2]In our experiment, we have arbitrarily chosen $R_1 = 0x5ed50e23$, $R_2 = 0x1b75e0d1$.

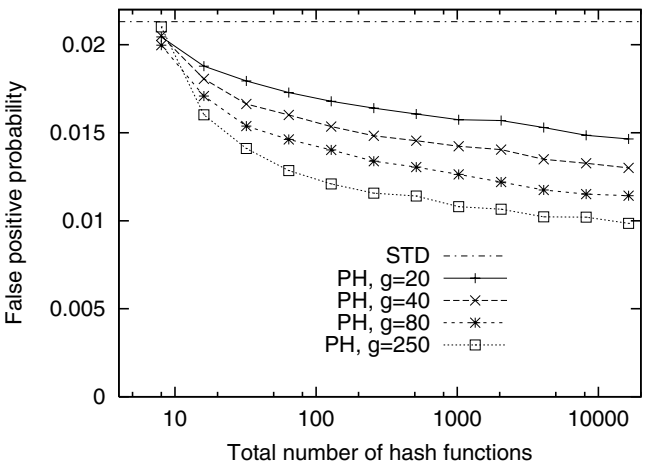

Figure 10: Impact of total number of hash functions H on false positive probability

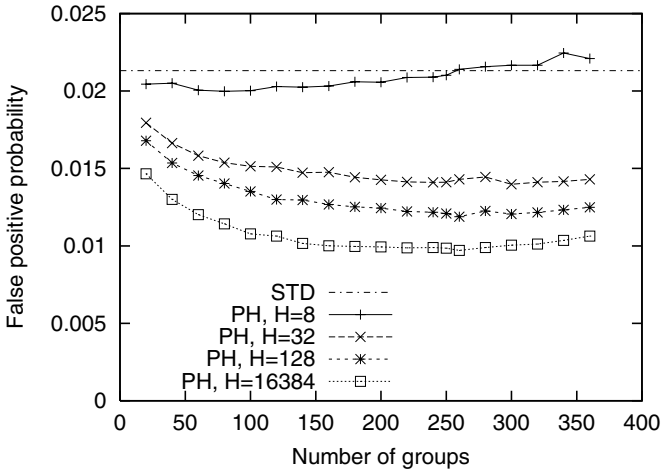

Figure 11: Impact of number of groups g on false positive probability

and H imply more bits to be stored for the selected hash functions, and hence leaving fewer bits for actual hashing. In this experiment, we hope to find the proper tradeoff for both parameters to minimize false positive probability. The results that we present here correspond to $n = 10000$ and $m = 80000$, although our observation is commonly applicable to other settings. The only difference is that the optimal H and g values may differ across different n and m values.

Figure 10 shows the impact of H on false positive probability p_a. Note that x-axle is shown in log scale. We observe that p_a decreases with increase in H. The reduction in p_a becomes less as H increases. Comparing Figure 10 with Figure 7, we find that indeed the results from the simulation agrees well with the previous numerical analysis if we ignore the detailed difference caused by difference in settings. Note that number of trials in Figure 7 is determined by H in Figure 10 since more hash functions implies more possible combinations that can be explored.

Figure 11 shows the change of p_a with increase in g. We observe that the optimal value of g may differ for different H values. $H = 16384$ and $g = 250$ give the lowest p_a. Overall, p_a seems to be much more sensitive to change in g than in H. This is consistent with the fact that PH takes $g \cdot k \cdot log_2 H$ bits out of the Bloom filter bit vector for identifying the hash functions of all groups. This cost is a linear function of g but log function of H. Comparing Figure 11 with Figure 6, we find that unlike in Figure 6, here increase in g does not always lead to decrease in false positive probability. This is because the previous analysis has ignored the impact of memory cost.

6.2 Comparisons with standard and PTC Bloom filters

We compare the performance of PH filter with both standard and PTC filters. For convenience, we use the same m and n settings as those used in [3]. We take the best results of PTC from [3] and the optimal settings for PH and compare their performance. Table 1 lists the false positive probabilities of three different approaches when $\frac{m}{n}$ is 8, 16, and 32. n is set to be 10000 in this experiment, but the result seems to be consistent for other n values. The settings of PH filter are shown in Table 2.

Methods	$\frac{m}{n} = 8$	$\frac{m}{n} = 16$	$\frac{m}{n} = 32$
Standard filter	0.0216	4.59×10^{-4}	2.11×10^{-7}
PTC filter	0.0122	1.58×10^{-4}	3.53×10^{-8}
PH filter	0.0098	1.13×10^{-4}	1.66×10^{-8}

Table 1: False positive probability of standard, PTC, and PH Bloom filters

We find that PH filter performs the best in all cases. The false positive probability of the PH filter is 50% to an order of magnitude smaller than the standard Bloom filter, and 20% to 50% smaller than the PTC filter. Another significant advantage of PH over PTC is that for each new arrival, PTC requires to hash $c \cdot k$ times, where c is the number of choices and k is the number hash functions; while PH only requires to hash $k + 1$ times.

PH parameter	$\frac{m}{n} = 8$	$\frac{m}{n} = 16$	$\frac{m}{n} = 32$
g	250	250	180
H	16384	16384	16384
k	6	12	24

Table 2: Configuration parameters of PH Bloom filters

6.3 Experiments with real IP trace

In this section, we further validate our results by applying the CRC based hash functions to the content of real IP packets. We use the trace that contains complete packet payload

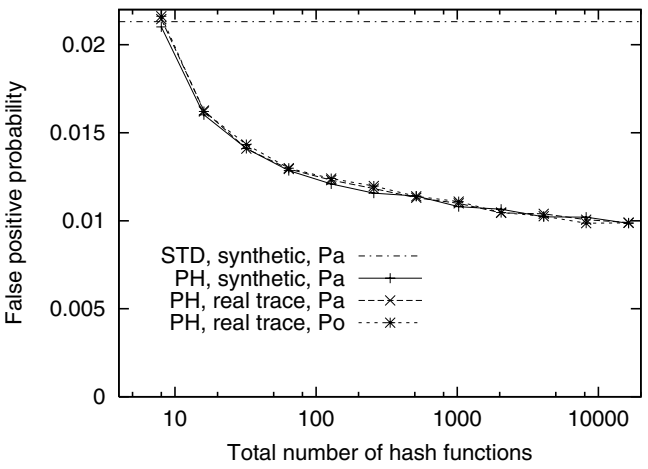

Figure 12: Comparison of experiments with synthetic traces and real IP packet traces: $\frac{m}{n} = 8$

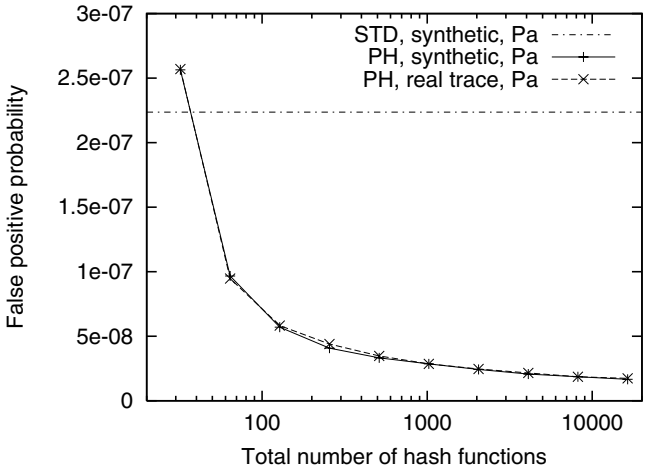

Figure 14: Comparison of experiments with synthetic traces and real IP packet traces: $\frac{m}{n} = 32$

information[3]. This trace contains incoming anonymous FTP connections to 320 public FTP servers at Lawrence Berkeley National Laboratory during a 10-day period.

Figures 12, 13, and 14 show the comparison between the case of using synthetic trace and random hashing, and the case of using real trace and CRC hash function. Standard Bloom filter is also shown as reference. Note that all figures show both p_a and p_o except Figure 14. This is because the false positive probability is very low in Figure 14, and the trace does not contain enough packets to generate a meaningful result.

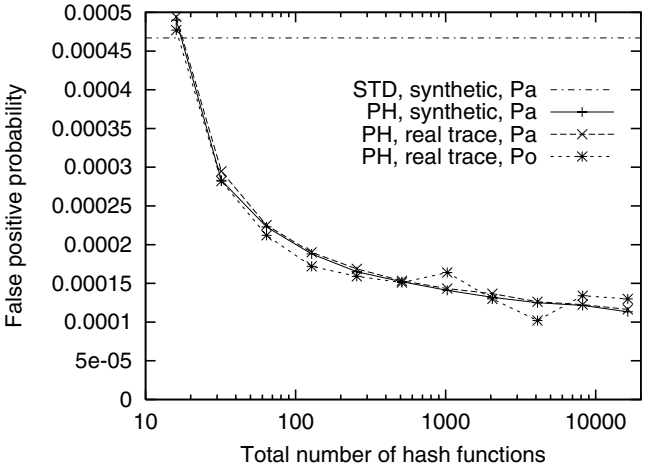

Figure 13: Comparison of experiments with synthetic traces and real IP packet traces: $\frac{m}{n} = 16$

Overall, we find that synthetic simulation agrees with trace-based simulation very well. The small noise in p_o in Figure 13 when the false positive probability is very low is again caused by not having enough packets in the trace to estimate such rare events. The strong agreement between the simulations indicates that in practice, we can use the CRC based hash functions to implement the PH approach and get a much more accurate Bloom filter than the standard one.

[3]http://www-nrg.ee.lbl.gov/anonymized-traces.html

7. CONCLUSION

In this paper, we have proposed a novel approach to improve accuracy of the Bloom filter by tailoring the hash functions. The approach works by partitioning the keys into multiple groups and selecting proper combinations of hash functions for each group, so that the overall fill factor of the filter is reduced. We have shown through both analysis and experiments that the Partitioned Hashing approach can significantly the false positive probability when compared to the standard Bloom filter as well as other improved Bloom filter scheme such as Power of Two Choices filter [3]. We have further validated our results by using real CRC based hash functions and real IP packet traces.

Further research needs to be done to investigate how to calculate the optimal configuration parameters such as number of groups g and number of hash functions H. Although we have studied the tradeoffs of various settings of such parameters, we currently can only derive the optimal values through experiments and observations. Secondly, it is very likely that the simple greedy algorithm for searching hash functions for groups only returns a local optimal value. It will be interesting to find out how far this value is from the true optimal value, and whether more effective algorithms can be developed to further improve the performance. Thirdly, we have not exhausted all dimensions of design. For instance, different groups may not need to use the same number of hash functions. It remains to be seen whether one can reduce false positive probability by selectively removing hash functions from certain groups. Finally, the current

approach works best when the key set is fixed. Periodic re-optimization may be necessary when the key set dynamically changes. More studies need to be done to properly handle such cases.

8. REFERENCES

[1] B. H. Bloom, "Space/time tradeoffs in hash coding with allowable errors", *Communications of the ACM 13:7 (1970), 422-426.*

[2] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better Bloom filter", *ESA 2006.*

[3] S. Lumetta and M. Mitzenmacher, "Using the power of two choices to improve Bloom filters", *Preprint version available at http://www.eecs.harvard.edu/ michaelm.*

[4] A. Broder and M. Mitzenmacher, "Network applications fo Bloom filters: a survey", *Internet Mathematics, vol. 1. no. 4, pp. 485-509*, 2005.

[5] R. Motwani and P. Raghavan, "Randomized algorithms", *Cambridge University Press*, August, 1995.

[6] Robert Sedgewick, "Algorithms in C", *Addison-Wesley Professional*, August, 2001.

[7] A. Ostlin and R. Pagh, "Uniform hashing in constant time and linear space", *Proceedings of STOC 2003, ACM.*

[8] A. Pagh, R. Pagh, and S. Rao, "An optimal Bloom filter replacement", *SODA 2005.*

[9] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables", *SODA 2004.*

[10] C. Estan and G. Varghese, "New directions in traffic measurement and accounting", *Proceedings of ACM SIGCOMM Conference*, 2002.

[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* 8:3 (2000), 281-293.

[12] E. H. Spafford, "Opus: preventing weak password choices", *Computer and Security* 11 (1992), 273-278.

[13] U. Manber and S. Wu, "An algorithm for approximate membership checking with application to password security", *Information Processing Letters* 50 (1994), 191-197.

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom filters: from approximate membership checks to approximate state machines", *Proceedings of ACM SIGCOMM 2006.*