IEICE TRANS. INF. & SYST., VOL.E91–D, NO.5 MAY 2008

PAPER Special Section on Information and Communication System Security

Adaptive Bloom Filter: A Space-Efficient Counting Algorithm for Unpredictable Network Traffic

Yoshihide MATSUMOTO[†a)], Hiroaki HAZEYAMA[†b)], *Nonmembers, and* Youki KADOBAYASHI[†c)], *Member*

SUMMARY The Bloom Filter (BF), a space-and-time-efficient hash-coding method, is used as one of the fundamental modules in several network processing algorithms and applications such as route lookups, cache hits, packet classification, per-flow state management or network monitoring. BF is a simple space-efficient randomized data structure used to represent a data set in order to support membership queries. However, BF generates false positives, and cannot count the number of distinct elements. A counting Bloom Filter (CBF) can count the number of distinct elements, but CBF needs more space than BF. We propose an alternative data structure of CBF, and we called this structure an Adaptive Bloom Filter (ABF). Although ABF uses the same-sized bit-vector used in BF, the number of hash functions employed by ABF is dynamically changed to record the number of appearances of a each key element. Considering the hash collisions, the multiplicity of a each key element on ABF can be estimated from the number of hash functions used to decode the membership of the each key element. Although ABF can realize the same functionality as CBF, ABF requires the same memory size as BF. We describe the construction of ABF and IABF (Improved ABF), and provide a mathematical analysis and simulation using *Zipf*'s distribution. Finally, we show that ABF can be used for an unpredictable data set such as real network traffic.

key words: *Bloom Filter, counting, burst traffic*

1. Introduction

The Bloom Filter (BF), a space-and-time-efficient data structure and hash coding method, is used as a high-speed membership-testing module in several network processing algorithms and applications, such as route lookup [9], cache sharing [5], key word search [6], per-flow traffic measurement [7], path tracking of a single spoofed packet [8], and deep packet inspection [4].

In BF, the bit-vector is overwritten with each hash key, so that the BF allows false positives and cannot count the number of distinct hash keys. Several extensions of BF [2], [3], [5] have been proposed to count distinct keys, and these extensions need to estimate the memory space to count targets in advance. When those BF extensions are employed for real-time packet classification or traffic analysis in a high speed network, reallocating memory dynamically is difficult due to limited memory space on the hardware or limited time of calculation in the wire rate. If numerous same packets, such as DoS/DDoS attacks, arrive over and over again, the counter bit overflows easily. Allocating a huge amount of memory for flush clouds or DDoS packets is expensive.

In this paper, we present an Adaptive Bloom Filter (ABF) which is a space-efficient hash coding with a counting function. The key idea of ABF is to change the number of hash functions k along with the number of appearances of each key of an element in a certain time period. For each distinct key element, ABF sets bits indicated by each k hash function to 1 in the same way as BF. When all k bit positions have already been 1, a registered key element arrives again, ABF calculates the bit position by another hash function. In a query of membership tests, the additional number of hash functions N indicates the number of appearances of each entry. ABF does not need to prepare extra memory space to count the multiplicity of a each element. ABF is useful for counting a packet in a network where the maximum rate of packet per second is unpredictable.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 explains ABF in detail. In Sect. 4 an improved ABF is explained. In Sect. 5, we show the results of the simulations. Section 6 discusses applying ABF to IP Traceback, and the present study is summarized in Sect. 7.

2. Related Work

2.1 Bloom Filters

The Bloom Filter (BF) [1] is used as one of the fundamental modules in several network processing algorithms. BF is a simple space-and-time-efficient data structure used to represent a set in order to support membership queries. Scoping on only testing membership, BF reduces the required memory space to record membership, although false-positives occurs. Because of its scope, BF does not have a counting or deleting function.

In Fig. 1, four hash values generated by element a_n set bit-vector B of length m. A set H = $H_1, H_2, \ldots, H_k$ of k independent hash functions, each with range 1,...,m and the bits at positions $H_k(a_n)$ in B are set to 1. To query for an element, the element must be fed to each of the k hash functions to get k array positions. However, if any of the bits at these positions are 0, the element is not in the set.

False-positives occur because all the elements fill the bit of a shared bit-vector. For given m and n, the value of k (the number of hash functions) leads to minimized false-positive probability (Fig. 2). However, false-negatives never

Manuscript received July 31, 2007.
Manuscript revised December 13, 2007.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
a) E-mail: yoshi-ma@is.naist.jp
b) E-mail: hiroa-ha@is.naist.jp
c) E-mail: youki-k@is.naist.jp

DOI: 10.1093/ietisy/e91–d.5.1292

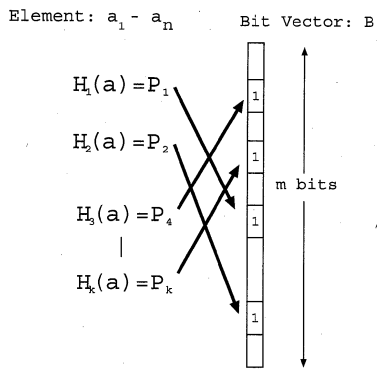


Fig. 1 Bloom Filter with four hash functions.

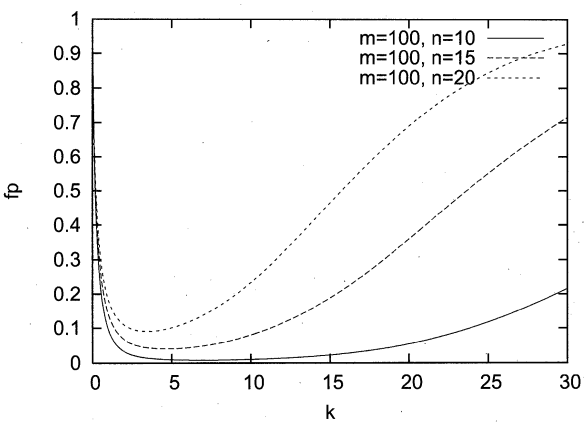


Fig. 2 False positive probability.

occur unless a delete operation is given.

The probability that a certain bit is not set to 1 by a hash function in m bits memory during the insertion of an element is then

$$1 - \frac{1}{m}. \tag{1}$$

The probability that a certain bit is still 0 when we insert n element using k hash function is then

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} \tag{2}$$

and the probability that a certain bit is 1 is

$$p = 1 - e^{-\frac{kn}{m}} \tag{3}$$

The probability of all positions of k function being 1 is

$$f_{pBF} \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{4}$$

$p = 1/2$ is the most space-efficient value. Thus the minimum false positive probability for given values of m and n is

$$f_{pBF} \approx \left(\frac{1}{2}\right)^k \approx 0.6185^{m/n} \tag{5}$$

2.2 Counting Bloom Filters

A counting Bloom Filter (CBF), which is one of the extensions of BF, provides a increment and decrement operation. The bits of the array which are replaced are extended from a single bit, to become a c-bit counter. Each counter represents the number of times that hashed values by k hash functions located each bit position on the filter. CBF can also delete the element by decreasing the counters.

Instead of counting and deleting functions, CBF has several problems. First, an overflow of c-bit counters easily occurs when skewed data arrives. Second, CBF requires a large memory space for c counter bits. The required memory space of CBF is c times the required memory of BF. If the input element is highly skewed, CBF wastes memory space because almost all bits and their counters are not used to record elements.

3. Adaptive Bloom Filters

The Adaptive Bloom Filter (ABF) uses only a single bit-vector as with a Bloom Filter. ABF can estimate the duplication of each key in the same way as CBF.

ABF changes the number of hash functions dynamically. The basic idea of counting hash collisions on ABF is as follows. Basically, ABF works as BF, that is, ABF sets each bit position indicated by k hash functions to 1. When all k bit positions have already been set to 1 to record one element, ABF calculates a hash value by using an additional $k+1$'th hash function. Furthermore, if the bit located by the $k+1$ th function has already been 1, ABF iteratively calculates by another hash function until the bit indicated by $k+N+1$'th hash function is 0. In the query of a membership test, the number of additional hash functions N represents the number of appearances of each key element if the rate of hash collisions among distinct hash functions approximates zero. Algorithm 1 shows the insertion algorithm of ABF. The querying of membership on ABF is described in Algorithm 2. Figure 3 illustrates the differences of the insertion sequence among BF, CBF and ABF. Instead of the overhead on the querying, ABF saves the memory space for the c-bit counter of CBF.

3.1 Algorithm for Insertion

In this section, we describe the insert algorithm on ABF (Algorithm 1). All of these algorithms are simple to implement.

BF usually uses fixed k hash functions, but ABF uses $k+N+1$ independent hash functions. In algorithm 1, check whether the bit located H_{k+N} is set to 1 while being increment N. The bit located $H_{(k+N+1)}$ is set to 1.

3.2 Algorithm for a Query

In algorithm 2, we first check whether all of H_k is set to 1 or not. This operation is the same as BF. Next, we check

Algorithm 1 ABF: Algorithm for Insertion

Require: B, the bit-vector and v, input element
Ensure: N, number of additional hash function
1: **if** all $B[H_{1..k}(v)] = 1$ **then**
2: $N \Leftarrow 1$
3: **while** $B[H_{k+N}(v)] = 1$ **do**
4: $N \Leftarrow N + 1$
5: **end while**
6: $B[H_{k+N}(v)] = 1$
7: **else**
8: all $B[H_{1..k}(v)] = 1$
9: **end if**

Algorithm 2 ABF: Algorithm for a Query

Require: B, the bit-vector and v, input element
Ensure: N, number of additional hash function
1: **if** all $B[H_{1..k}(v)] = 1$ **then**
2: $N \Leftarrow 1$
3: **while** $B[H_{k+N}(v)] = 1$ **do**
4: $N \Leftarrow N + 1$
5: **end while**
6: **return** N
7: **end if**

whether all of H_{k+N} is set to 1 or not. We get parameter N, which means that the number of additional hash function within the bit-vector is set in the same way as the insert Algorithm (Algorithm 1). In other words, N represent the quantity of elements whose accuracy depends on occupancy rate p.

3.3 Estimating the Number of Appearances of Each Element

We get N as iteration using the Algorithm 2 for query. But the N includes the quantity of elements and errors because the bit-vector is overridden by another element randomly. We should estimate the quantity of elements $\hat{N}$ using N.

Figure 4 presents an occupancy rate p (Eq. (3)) that represents one bit set to 1 when the n'th element arrives on BF. $p = 0.5$ is the most efficient equation for saving space in BF.

$$p_{max} = 1 - e^{-\frac{kn}{m}} = \frac{1}{2} \tag{6}$$

We assume a_n is an n'th input element, and we define D is a set of elements which value is v in a bit-vector.

$$D = \{a_n | a_n = v\} \tag{7}$$

The more often the occupancy rate p is higher, the greater the N. If the n'th element adds 1 bit located at H_{k+N}, H_{k+N+1} bit is already set to 1 with high probability. Therefore, the N depends on the occupancy rate p_n when n'th element arrives, and N increases with $\frac{1}{1-p_n}$ bits in the bit-vector.

$$N = \sum^{a_n \in D} \frac{1}{1 - p_n} \tag{8}$$

Our simple technique estimates the number of keys

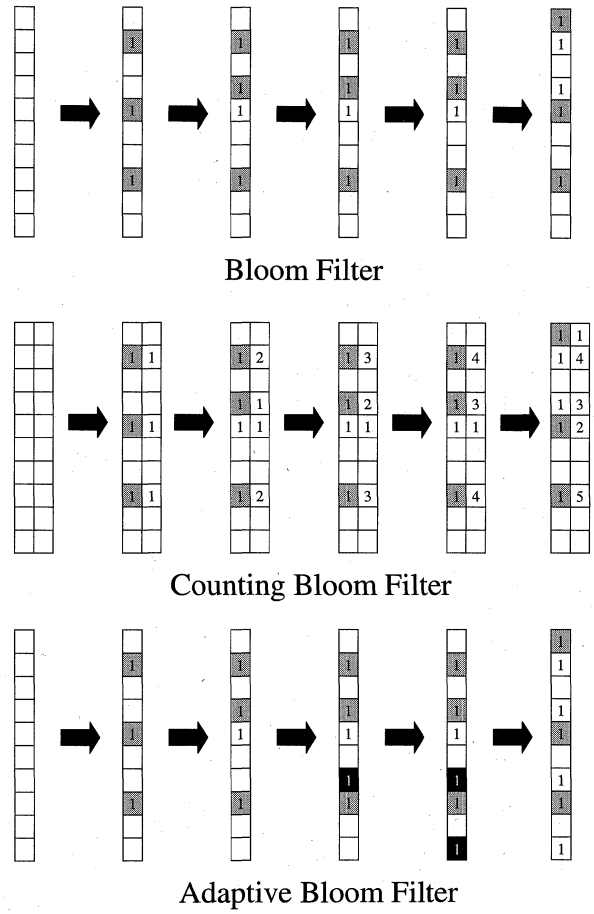


Fig. 3 Insert method of BF, CBF and ABF.

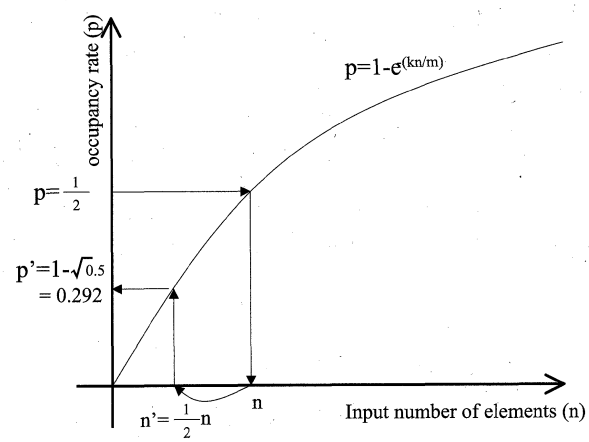


Fig. 4 Simple estimation using average: The occupancy p that is the probability one bit is set to 1 with in the bit-vector.

needed to get an average p. We get p_{avg} using half of input number of elements.

$$n_{avg} = \frac{1}{2}n \tag{9}$$

$$p_{avg} = 1 - e^{-\frac{kn_{avg}}{m}} = 1 - e^{-\frac{0.5kn}{m}} = 1 - e^{-\frac{1}{2}ln2}$$

$$= 1 - \sqrt{0.5} \approx 0.292 \quad (10)$$

We define $\hat{N}$ as the estimated quantity of each elements. We simply use $p_{avg} = 1 - \sqrt{0.5}$ for the estimation.

$$N \approx \frac{1}{1 - p_{avg}} \cdot \hat{N} = \frac{1}{\sqrt{0.5}} \cdot \hat{N}$$

We get $\hat{N}$ as the estimated counter of each element.

$$\hat{N} \approx \sqrt{0.5} \cdot N \approx 0.707 \cdot N \quad (11)$$

However, accuracy of this simple technique depends on the order of arrival.

4. Improved Adaptive Bloom Filter (IABF)

The weak point of ABF is that the accuracy of $\hat{N}$ changes depending on the order of arrival pattern. We introduce IABF, that IABF inserts 1 or 2 bits ($z_{add} \subseteq 1, 2$) stochastically each time although ABF inserts only 1 bit. The number of additional hash functions N increases with $\frac{1}{1-p_n}$ bits in ABF. But in IABF, N increases 2 bits instead of $\frac{1}{1-p_n}$.

z_{add} is calculated is as follows;

$$\frac{1}{1 - p_n} z_{add} = 2$$

$$z_{add} = 2(1 - p_n)$$

As a result, N increases 2 bits all of the time, $\hat{N}$ is

$$N = \sum_{n=1}^{\hat{N}} \frac{1}{1 - p_n} z_{add} = \sum_{n=1}^{\hat{N}} 2 = 2\hat{N} \quad (12)$$

$$\hat{N} = \frac{1}{2} \cdot N = 0.5 \cdot N \quad (13)$$

4.1 The Method of Inserting a Decimal bit

Add z_{add} bit is difficult because z_{add} bits is represented by a decimal number. We add z_{add} bits stochastically using random number generator. IABF already insert 1 bit such as ABF. Next we need insert $z_{add} - 1$ bit stochastically. Algorithm 3 shows the insertion algorithm of IABF. The *rand*() generate a decimal number ($0 \leq rand() < 1$).

Algorithm 3 IABF: Algorithm for Insertion

Require: B, the bit-vector v, the input element and p_n, the occupancy rate
Ensure: N, number of additional hash function

```
if all B[H_{1..k}(v)] = 1 then
    N ⇐ 1
    while B[H_{k+N}(v)] = 1 do
        N ⇐ N + 1
    end while
    B[H_{k+N}(v)] = 1
    if rand() ≤ 2(1-p_n) − 1 then
        B[H_{k+N+1}(v)] = 1
    end if
else
    all B[H_{1..k}(v)] = 1
end if
```

5. Evaluation

5.1 False Positive

When BF uses k hash functions and m sized memory with n unique entries, then, the false positive of BF f_{pBF} is defined as

$$f_{pBF} \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (14)$$

To define the false positive rate of CBF, ABF or IABF, we have to consider the collision elements because of the counting elements.

Here, we define collision rate α. α as collision elements / total elements. For example, if the input elements are {1,2,2,3,3}, the unique elements are {1,2,3}, and the collision elements are {2, 3}. α is {2,3} / {1,2,2,3,3} = 2/5 = 0.4. The maximum number of counter is collision elements / unique elements = $\frac{\alpha}{1-\alpha}$.

Then, the required bit size for counter bits c with no skew of input elements is

$$c = \left\lceil \log_2 \lceil \frac{\alpha}{1 - \alpha} \rceil \right\rceil \quad (15)$$

5.1.1 False-Positive of CBF

If the bit size for the counter of CBF is c and the total memory size is m, CBF can use only m/c-sized memory space to register n elements. Therefore, the false positive of CBF is defined as:

$$f_{pCBF} \approx \left(1 - e^{-\frac{knc}{m}}\right)^k. \quad (16)$$

5.1.2 False-Positive of ABF

Next, we try to define the false positive of ABF. ABF uses the total memory space m for registering unique elements such as BF. Different from BF, ABF also uses the same bit-vector to count collision elements. The false-positive of ABF f_{pABF} is influenced by the collision ratio α and the order of arrival. We define the size of amount bit $q = n(1 - \alpha)$, which is filled by unique elements, and the size of amount bit $r = n\alpha$ which is filled by collision elements.

We assume T_n is the probability that the n'th element is not set to 1 during the insertion, The mT_{n-1} is not set to 1 in m, which must be decreased when the collision element arrives. We also assume that $T_0 = 1$.

$$T_n = \begin{cases} \prod^q (1 - \frac{1}{m})^k & (unique\ elements) \\ \prod^r (1 - \frac{1}{mT_{n-1}}) & (collision\ elements) \end{cases}$$

$$f_{pABF} = (1 - T_n)^k$$

Figure 5 presents a false positive probability with BF, CBF and ABF. ABF is more efficient than CBF where the collision rate is lower. The reason the collision rate is lower is because ABF consumes n bit when n unique elements insert the bit-vector. On the other hand, CBF uses $log_2(n)$ bits for the counter.

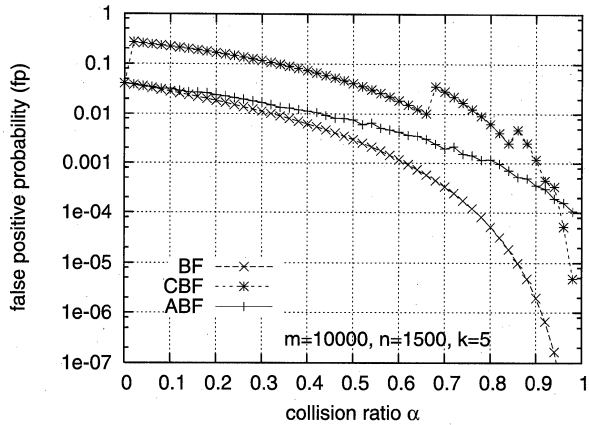


Fig. 5 False-positive probability (f_p) with collision ratio (α).

Table 1 The order of space.

Type	Required Memory	Counter Size
BF	2^l	N/A
CBF	$2^l c$	2^c
ABF	2^l	$\lfloor \frac{(2^l - k_{min})}{2} \rfloor$
IABF	2^l	$\lfloor \frac{(2^l - k_{min})}{4} \rfloor$

5.1.3 False-Positive of IABF

Although ABF inserts 1 bit per input element, IABF inserts $2(1 - p_n)$ bits. The false-positive of IABF is as follows:

$$T_n = \begin{cases} \prod^{q}(1 - \frac{1}{m})^k & \text{(unique elements)} \\ \prod^{r}(1 - \frac{1}{mT_{n-1}})^{2(1-p_n)} & \text{(collision elements)} \end{cases}$$

$$f_{pIABF} = (1 - T_n)^k$$

5.2 Required Memory Size

Here, we describe the required memory space by the bit length of the output hash value. Because of a characteristic of BF, the required memory for the bit-vector is determined by the bit length of the output hash value.

Table. 1 compares BF, CBF, ABF and IABF regarding the required memory size and capacity of the counting function.

First, we try to describe the required memory size by the bit length of the output hash value by each hash function l. BF requires 2^l-sized memory for storing membership. CBF has to prepare the extra memory space for counting in advance. When the size of the each counter bit is c, the size of required memory for CBF is $2^l c$. Both ABF and IABF require 2^l-sized memory while ABF uses the memory for recording membership and counting the collision elements.

Next, we define the number of appearances of each element A_I for a unique input element I, and $max(A_I)$ is the maximum of A_I. $max(A_I)$ of CBF $max(A_I)_{CBF}$ depends on the size of the counter bit, hence,

$$max(A_I)_{CBF} = 2^c$$

ABF records one extra bit to count the number of elements. ABF is allowed to count elements until the occupancy of the bit-vector is less than 50%, therefore, if the minimum set of the hash functions is defined as k_{min}, then,

$$max(A_I)_{ABF} = \left\lfloor \frac{(2^l - k_{min})}{2} \right\rfloor$$

IABF records 1 - 2 more bits to count the same key element, hence,

$$max(A_I)_{IABF} = \left\lfloor \frac{(2^l - k_{min})}{4} \right\rfloor$$

5.3 Capacity of the Filter

Here, we compare the capacity of each filter algorithm to record distinct elements.

The capacity of recording membership n on BF depends on bit-vector size $m = 2^l$ and the allowed false positive rate f_{pBF}. By solving equation 1 for the memory efficiency factor (n/m) and differentiating with respect to k, it is easy to check that optimal memory efficiency is reached when $k = \log(1/f_{pBF})$. The memory requirement of such a table can be determined easily by substituting f_{pBF} back into equation 1 (observe $f_{pBF}^{1/k} = 1/2$):

$$m = -n_{BF} \frac{\log(1/f_{pBF})}{\ln(1/2)}$$

Therefore, if the allowed false positive rate f_{pBF} is given, then BF's capacity n_{BF} is

$$n_{BF} = -m \frac{\ln(1/2)}{\log(1/f_{pBF})}$$

Considering CBF's capacity, the total memory space m is consumed to registering distinct entry and counting each entries. The memory space used to register distinct elements is m/c (c is the size of counter bit). Hence, CBF's capacity is n_{CBF} where the allowed false-positive rate is f_{pCBF},

$$n_{CBF} = -m \frac{\ln(1/2)}{c \cdot \log(1/f_{pCBF})}$$

ABF's capacity of registering membership is more complex because ABF changes k dynamically and f_{pABF} is decided by degree of skewness and the order of arrival. If all the elements are unique, f_{pABF} is the same as f_{pBF}. However, almost all the collision elements lead to worst capacity. The capacity of ABF n_{ABF_max} is

$$\begin{aligned} n_{ABF\_max} &= \frac{m}{2} - \left(1 - \left(1 - \frac{1}{m}\right)^k\right) \cdot m \\ &= \frac{m}{2} - (1 - e^{k/m}) \cdot m \\ &= \left(e^{k/m} - \frac{1}{2}\right) \cdot m \end{aligned} \tag{17}$$

Table 2 The order of calculation.

Type	Insert	Query	Query the counter
BF	$O(k)$	$O(k)$	N/A
CBF	$O(k)$	$O(k)$	$O(k)$
ABF	$O(k + A_I)$	$O(k)$	$O(k + 1/\sqrt{0.5A_I})$
IABF	$O(k + A_I)$	$O(k)$	$O(k + 2A_I)$

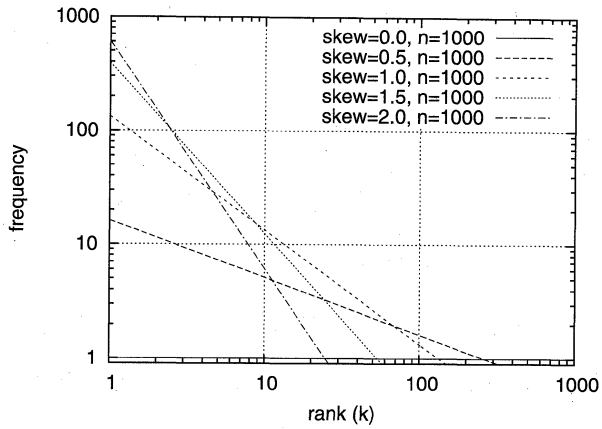


Fig. 6 Zipf's distribution for evaluation.

5.4 Calculation Overhead

Here, we compare the calculation overhead among BF, CBF, ABF and IABF. Table. 2 compares the order of calculation of each filter. ABF needs iteration of A_I in addition to the calculation of BF.

5.5 Characteristics of the Accuracy of Counting

In this section, we compare the accuracy of the estimation on the counted number in each algorithm. We compare CBF, ABF and IABF with *Zipf*'s distribution in order to evaluate the feature of input elements with the varying deviations (Fig. 6). The parameter *skew* (θ) represent the characteristic of the deviation of *Zipf*'s distribution. For example, if the skew = 0, the number of each collision element is distributed equally.

At first, we tried to characterize the distribution of differences between the estimated counter and the actual counter. We evaluated the case where n is 1000, the total memory size m of all algorithm is the same, all algorithms use the same parameter $k = 3$, and $F_p = 0.125$. And another parameter of CBF or ABF is chosen with most efficiently on each algorithm. The deviation of ABF is figured in Fig. 7, and CBF, in Fig. 8. We assume that the correct value is located at 0. The almost all estimated counter was in a -5 to 5 with varying skew θ.

Next, we evaluated the error rate of the counting function on each filter algorithm. We implemented CBF, ABF and IABF, and measured the accuracy of the estimated counter for each element.

We defined the parameters as follows:

$\hat{f}_i$ is the estimated times of the duplication, f_i is the actual counter for the i'th element in a set of unique elements v.

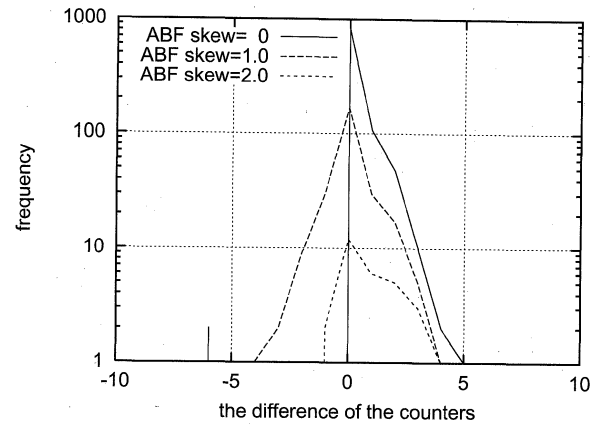


Fig. 7 Deviation of accuracy of ABF with varying skew (θ), the number of input elements (n) is 1000.

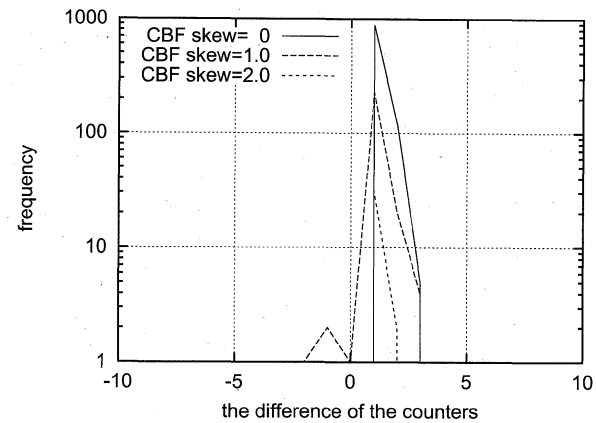


Fig. 8 Deviation of accuracy of CBF with varying skew (θ), the number of input elements (n) is 1000.

We measured the mean squared additive error, which is calculated by

$$E_{add} = \sqrt{\frac{\sum_{i \in v}(\hat{f}_i - f_i)^2}{n}} \tag{18}$$

Figure 9 shows the error rate of the counter of each filter algorithm along with *Zipf*'s distribution in Fig. 6. CBF shows a high error rate in the middle skew because of the counter overflow. At the lower size of the skew, CBF's counter bit is evenly-distributed, so that utilization of the space is efficient. In the high size of skew, CBF could allocate almost all memory which is used as the counter.

Next, we evaluate the accuracy of counting function in the skewed arrival of elements. In Fig. 10, the order of the data set was sorted. At the higher level of the skew, the error rate of ABF was much larger than that in Fig. 9, but IABF still kept a lower error rate.

In sum, the accuracy of IABF is higher and more stable than CBF or ABF.

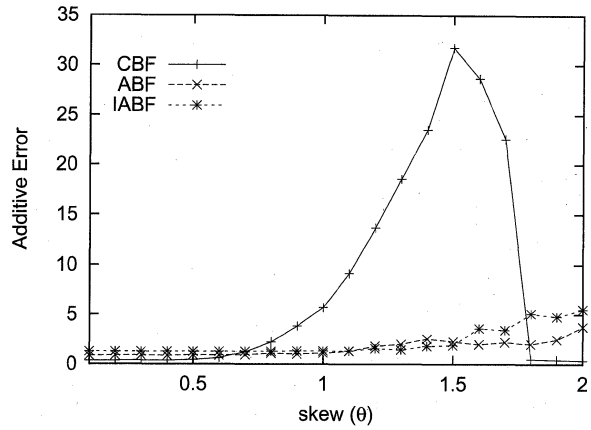


Fig. 9 Accuracy of CBF, ABF and IABF for zipf's distribution which is order by at random.

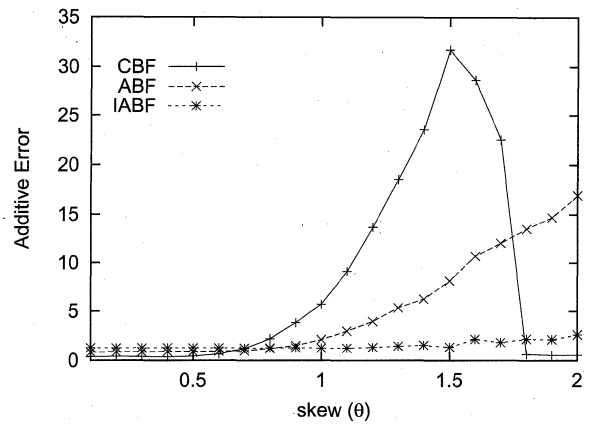


Fig. 10 Accuracy of CBF, ABF and IABF for zipf's distribution which is order by distinct element.

6. Discussion

6.1 ABF Support for Biasing and Unpredictable Network Traffic

In real-time network traffic analysis, reallocating the memory structure of CBF in real time is difficult. Therefore, we need to estimate the memory space that is $C \cdot m$ (the counter bit, the memory size) bits for counting in advance. Furthermore, if the same packet flow arrives over and over again, such as in DoS/DDoS attacks, the counter bit easily overflows.

ABF does not have special space for the counter bit because in ABF has the bit for the counting into the bit-vector. Hence, ABF insertion could occur with no overflow and no preparation of the counting space in advance. In other words, we do not need to predict the rate of duplication, hash collisions and / or deviations of measured traffic.

6.2 The Delete Function on ABF

ABF does not support deletion and updating of the counter, because if one tries to remove the bit corresponding to the hash value, the bit leads to a false negative and a degradation of accuracy regarding the counting.

To keep the accuracy of ABF, several ABFs such as in [8] must be rotated.

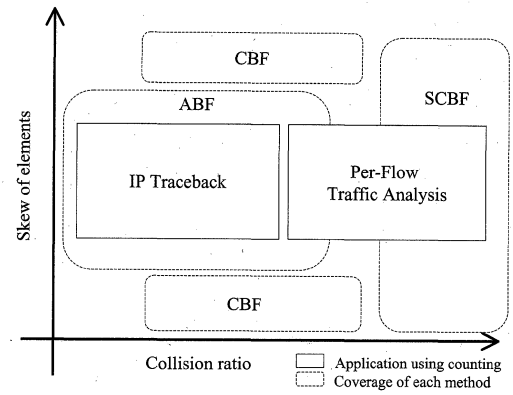


Fig. 11 Coverage of each method.

6.3 Applying ABF to Traffic Monitoring Application

The study of traffic monitoring is a fundamental technique of Network Security. ABF is useful for the network anomaly detection and monitoring unpredictable traffic such as DoS/DDoS attacks because of efficient data structures of ABF.

In Fig. 5 shows ABF is less false positive than CBF in $\alpha(collision\ rate) < 0.95$. ABF improves the false-positive ratio tenth compared to CBF without in high collision rate.

In Fig. 11, Space Code Bloom Filter (SCBF) [7] is suitable for counting high collision keys of network flow because SCBF uses multiple bit-vector and counts with percentages. Therefore, ABF could count network flow more detail as compared with SCBF in the low collision rate. Especially, ABF is suitable for monitoring a low collision ratio and / or a highly skewed traffic. The ABF can be widely applied on traffic monitoring and analysis.

For example, in the IP traceback [8], the collision ratio of hashed packet is approximately 0.00092% in the wide area and 0.139% on the LAN in the A 28-byte prefix. Although ABF needs more calculation cost compared with CBF, ABF improves the false-positive ratio tenth compared to CBF.

7. Conclusion

In this paper, we presented ABF as a new construction with the same functionality as the counting Bloom Filter (CBF) for network traffic analysis.

CBF has a special space for the counter bits, and overflow easily in real network traffic where collision rate changes dynamically and is unpredictable. On the contrary, ABF and IABF could count each element with no overflow and with no preparation of the counting space in advance.

The accuracy of ABF depends on the arrival pattern of

input elements. IABF can decrease the error rate on estimating the times of duplication on each input element by adding more two bits on the Bloom filter.

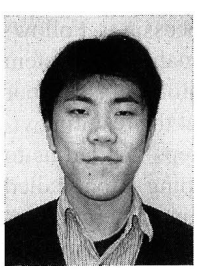
ABF and IABF are suitable for recording membership queries on the multiplicities of individual key elements with a skewed data set such as the unpredictable traffic data.

Acknowledgment

This research has been conducted as part of the National Institute of Information and Communications Technology (NICT) research program, "Research and Development of Traceback Technologies on the Internet."

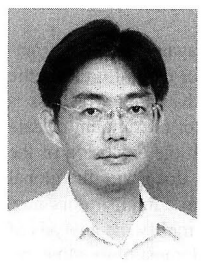
References

[1] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Commun. ACM, vol.13, no.7, pp.422–426, 1970.
[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," ESA, pp.684–695, 2006.
[3] S. Cohen and Y. Matias, "Spectral bloom filters," SIGMOD Conference, pp.241–252, 2003.
[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," IEEE Micro, vol.24, no.1, pp.52–61, Jan/Feb, 2004.
[5] L. Fan, P. Cao, J. Almeida, and A.Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," IEEE/ACM Trans. Netw., vol.8, no.3, pp.281–293, 2000.
[6] N. Jain, M. Dahlin, and R. Tewari, "Using bloom filters to refine Web search results," WebDB, pp.25–30, 2005.
[7] A. Kumar, J. Xu, J. Wang, O. Spatscheck, and L. Li, "Space-code bloom filter for efficient per-flow traffic measurement," Infocom, 2004, 2004.
[8] A.C. Snoeren, "Hash-based ip traceback," SIGCOMM '01: Proc. 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp.3–14, New York, NY, USA, ACM Press, 2001.
[9] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," ACM SIGCOMM, Philadelphia, PA, Aug. 2005.



Yoshihide Matsumoto received the M.E degree in computer science from the Nara Institute of Science and Technology, Japan, in 2003. His research interests include network security and network architecture for the Internet.

Hiroaki Hazeyama received his Ph.D degree in Engineering from Nara Institute of Science and Technology (NAIST), Japan, in 2006. He is currently an assistant professor in NAIST. His research interests include network operation, network security, and large-scale network testbed.



Youki Kadobayashi received his Ph.D. degree in computer science from Osaka University in 1997. He is currently an associate professor in the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His research interests include overlay networks, quality of services in the application-layer, and middleware security.