ID Bloom Filter: Achieving Faster Multi-Set Membership Query in Network Applications

Peng Liu[1*], Hao Wang[1*], Siang Gao[1], Tong Yang[1,2], Lei Zou[1], Lorna Uden[3], Xiaoming Li[1]
Peking University, China[1], Collaborative Innovation Center of High Performance Computing, NUDT, China[2],
School of Computing , Staffordshire University, UK[3]

***Abstract*—The problem of multi-set membership query plays a significant role in many network applications, including routers and firewalls. Answering multi-set membership query means telling whether an element belongs to the multi-set, and if yes, which particular set it belongs to. Most traditional solutions for multi-set membership query are based on Bloom filters. However, these solutions cannot achieve high accuracy and high speed at the same time when the memory is tight. To address this issue, this paper presents the ID Bloom Filter (IBF) and ID Bloom Filter with ones' Complement (IBFC). The key technique in IBF is mapping each element to k positions in a filter and directly recording its set ID at these positions. It has a small memory usage as well as a high processing speed. To achieve higher accuracy, we propose IBFC that records the set ID and its ones' complement together. The experimental results show that our IBF and IBFC are faster than the state-of-the-art while achieving a high accuracy.**

I. Introduction

A. Background and Motivations

Given s sets without intersection, the problem of multi-set membership query [1], [2] is to determine whether an incoming element belongs to the multi-set, and if yes, which set it belongs to. Multi-set membership query is a fundamental operation in many important network applications [3]–[6]. Online membership query is often operated on large datasets containing addresses, flow labels, signatures, *etc.* In a layer-2 switch, each destination MAC address is assigned to one unique port. When a packet is being forwarded, multi-set membership query needs to determine which port the packet should be forwarded to. The firewall also uses multi-set membership query to find out the suspicious level of each arrival packet, which is often determined by an intrusion detection system, and then takes certain follow-up actions.

Traditional exact-match data structures address the multi-set membership query problem, such as binary search tree and trie, store both keys and values, thus they suffer from huge memory and time overhead. When a low error rate is allowed, an alternative to solve this problem is the Bloom filter [7] [8], which is a compact probabilistic data structure that can tell whether an element exists in a single set with no false negative. Due to its small memory footprint and fast speed, most existing solutions to the problem of multi-set membership query are based on Bloom filters.

B. Limitations of Prior Art

A Bloom filter is a bit array with each bit set to `0` during initialization. When inserting an element, it uses k hash functions to map the element to k bits and set them to `1`. When querying an element, it hashes the element to k bits and checks the k corresponding bits. If they are all `1`, the element is claimed to be in the set; otherwise, the element does not belong to the set.

Combinatorial Bloom filter [9] is a typical Bloom filter based data structure for multi-set membership query, which contains one Bloom filter B and several groups of hash functions, and the i-th group has k hash functions $h_{i,1}(.), h_{i,2}(.), \ldots, h_{i,k}(.)$, associated with the i-th bit of the code of set ID. Each set ID is encoded into a l-bit constant weight code, which has a constant number of `1` bits for different set IDs. To insert an element e with set ID S_e, if the i-th $(1 \leqslant i \leqslant l)$ bit of S_e is `1`, e is mapped to k positions $B[h_{i,1}(e)], B[h_{i,2}(e)], \ldots, B[h_{i,k}(e)]$, and these bits are set to `1`. To query an element e, if the bitwise `AND` result of $B[h_{i,1}(e)], B[h_{i,2}(e)], \ldots, B[h_{i,k}(e)]$ is `1`, the i-th bit of the estimated code is set to `1`. For each query, Combinatorial Bloom filter needs to perform $k \cdot l$ memory accesses, thus its query speed is low, which will be improved by our algorithms.

C. Our Proposed Approach

In this paper, we propose the *ID Bloom Filter* (IBF) as well as its enhanced version *ID Bloom Filter with ones' Complement* (IBFC) to address the problem of multi-set membership query. The *key idea* of IBF is to record the set membership information directly in the mapped positions of each element.

The data structure of IBF and IBFC is composed of an array and k independent hash functions. The initialization of IBF and IBFC is to set all the bits in the array to `0`. IBF and IBFC both have two operations: insertion and query. We will first present a brief introduction to the operations of IBF, and then discuss the advantages of IBFC.

To insert an element e into IBF, we perform k hash computations and get k positions in the array. For each position, we do a bitwise `OR` operation between e's set ID and the binary string starting from the position. To query an element e, we

*Corresponding author: Tong Yang (Email: yangtongemail@gmail.com).
This work was done by Peng Liu, Hao Wang and Siang Gao under the guidance of their mentor: Tong Yang. This work is partially supported by Primary Research& Development Plan of China (2016YFB1000304), National Basic ResearchProgram of China (2014CB340405), NSFC (61672061), the OpenProject Funding of CAS Key Lab of Network Data Science and Technology,Institute of Computing Technology, Chinese Academy of Sciences.

also perform k hash computations and get k positions in the array. For each position, we retrieve a binary string, and then AND these k binary strings to get the estimated set ID for e. The limitations of IBF is that it may produce false results during queries because of hash collisions, and it cannot detect whether the query results are correct or not.

To address the limitations, we propose IBFC, which could detect the false results and return an Error during queries with a high possibility. IBFC improves the way of set ID encoding. To insert an element, we concatenate its set ID and the ones' complement of set ID, both in binary format, and then do bitwise OR operations at k positions to record the set ID information. To query the membership of an element, after computing the k positions, we retrieve k binary strings starting from these positions, and then AND these k binary strings to get an estimated binary string. Finally, we check the correctness of the above binary string, and give an estimated set ID if no Error is reported.

Key Contributions:

1) We proposed a novel Bloom filter and its enhanced version, namely IBF and IBFC, that directly store the set ID into each mapped positions of elements.
2) We have conducted extensive experiments with real datasets of flow traces, and the experimental results show that compared to the state-of-the-art, the processing speed of IBF and IBFC is greatly higher, and IBFC can detect more errors.

II. Related Work

Many works have been conducted addressing the problem of multi-set membership query, a large proportion of which are Bloom filter variants, such as Summary Cache [10], Coded Bloom Filter [11], [12], Bloomier Filter [13], [14] and Shifting Bloom Filter [15], [16]. Those variants are discussed briefly as follows.

Fan *et al.* proposed a scalable web cache sharing approach called Summary Cache, which contains s bloom filters where s is the number of distinct sets, and assigns each filter to a set. During insertion, it performs $k \cdot s$ hash computations to record an element, where k is the number of hash functions for each bloom filter. During query, it also performs $k \cdot s$ hash computations to query a set ID. This approach is straightforward yet inefficient in terms of speed, due to its large amount of memory accesses.

The Coded Bloom filter and Bloomier filter are both composed of multiple bloom filters. They both convert the set ID of an element to a binary string but use different ways to record the string. The Coded Bloom filter assigns one bloom filter for each bit of the string. It inserts an element into bloom filters according to the 1 bits of the string, and lookups element by checking all the bloom filters. This approach still has an insufficient query speed. The Bloomier filter contains two groups of Bloom filters and performs insertions for an element based on whether the bit of the string is 0 or 1. This approach has a higher query speed at the cost of not supporting dynamic insertion.

Shifting Bloom filter [16] is composed of one Bloom filter and k hash functions, and uses offsets to record the information of the set ID. To insert an element, it maps the element to k positions in the bit array, offsetting the k positions by a certain amount relevant to the set ID, and then set the k new positions to 1. To query an element, it performs k hash computations and checks s bits after these k positions, where s is the number of sets. However, it still has a large number of memory accesses.

III. Design of IBF and IBFC

In this section, we first describe the data structure, insertion and query operations of ID Bloom Filter (IBF), addressing its limitations, and then discuss the operations and advantages of an enhanced version, ID Bloom Filter with ones' Complement (IBFC). Table I summarizes the notations used in this paper.

TABLE I

Symbol	Description
B	a bit array
m	number of bits in B
$B[i]$	i-th bit in B
k	number of hash functions
$h_j(.)$	j-th hash function ($1 \leqslant j \leqslant k$)
s	number of different sets
l	maximum coding length of set ID
S_e	binary set ID of element e
C_e	ones' complement of S_e
V_e	bit string we record in B for element e
$V_e[i]$	i-th bit of V_e
$v_i(e)$	an l-bit string starting from $B[h_i(e)]$
w_e	number of 1 bits in S_e
n	number of distinct elements in all sets

A. ID Bloom Filter (IBF)

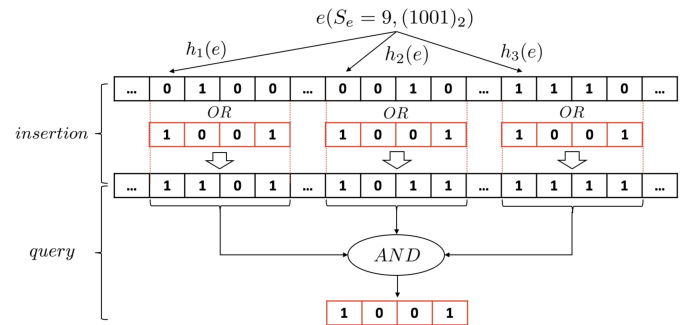

Fig. 1. Insertion and query of IBF.

Data structure: The data structure of IBF is an array B composed of m bits, the i-th of which is denoted as $B[i]$. IBF and IBFC are both associated with k independent hash functions, $h_1(.), h_2(.), \ldots, h_k(.)$, whose outputs are uniformly distributed in range $[1, m]$. The *initialization* operation of IBF and IBFC is to set all the bits to 0.

Insertion: To insert an element e with set ID S_e, first, e is mapped by k hash functions to k positions $B[h_1(e)], B[h_2(e)], \ldots, B[h_k(e)]$. Then, for each of the k positions $B[h_i(e)]$ $(1 \leqslant i \leqslant k)$, we perform a bitwise OR

operation between V_e and the l-bit binary string starting from $B[h_i(e)]$, denoted as $v_i(e)$, and store the result back into the l bit positions starting from $B[h_i(e)]$, where V_e is the binary format of set ID S_e. If $h_i(e) + l - 1 > m$, we store the extra bits into the first several bits of B. Specifically, we set $B[(h_i(e) + j)\%m] = B[(h_i(e) + j)\%m]$ OR $V_e[j]$, $1 \leqslant i \leqslant k$, $0 \leqslant j \leqslant l - 1$, where % meas modular operation.

For example, to insert an element e with set ID $S_e = 9$, as shown in Fig. 1, we first locate three positions by computing $h_1(e), h_2(e), h_3(e)$, and the 4-bit binary strings starting from these positions are $(0100)_2, (0010)_2, (1110)_2$. For each position, we do a bitwise OR operation between $(1001)_2$ and the 4-bit binary string, and record the result at the same position. Thus, the three binary strings are changed to $(1101)_2, (1011)_2, (1111)_2$.

Query: To query the membership of an element e, first we calculate k hash functions and locate k positions in B: $B[h_1(e)], B[h_2(e)], \ldots, B[h_k(e)]$. Then, we fetch k l-bit binary strings starting from the above k positions, denoted as $v_1(e), v_2(e), \ldots, v_k(e)$, and AND them together to get the result V_e. Specifically, $V_e = v_1(e)$ AND $v_2(e)$ AND ... AND $v_k(e)$. Finally we return V_e as the set ID.

For example, to query an element e, as shown in Fig. 1, we first locate three positions $h_1(e), h_2(e), h_3(e)$, and fetch 4-bit binary strings starting from these positions, which are $(1101)_2, (1011)_2, (1111)_2$. Then, we compute $(1101)_2$ AND $(1011)_2$ AND $(1111)_2$ and get the estimated set ID $(1001)_2 = 9$ for e.

Limitations: Based on the Bloom filter, a probabilistic data structure, IBF will suffer from *mis-classification* and *false positive*. Mis-classification means that an element from set i is reported to be in set j, where $i \neq j$. False positive means that an element that does not belong to the multi-set is reported to be in a certain set. Those two kinds of errors are a direct consequence of bit sharing in bloom filters, which means a bit can be set by different elements from different sets. Suppose we are inserting an element e whose set ID is 18, we insert $(10010)_2$ at k positions in B, $B[h_1(e)], B[h_2(e)], \ldots, B[h_k(e)]$. Unfortunately, we set $B[(h_i(e) + 1)\%m] = 1$ $(1 \leqslant i \leqslant k)$ during the insertion of other elements. Therefore, we get a wrong estimated set ID $(11010)_2$ instead of the right one $(10010)_2$ for e, which is a mis-classification. Because IBF cannot detect errors, mis-classification and false positive may occur using IBF.

B. IBF with ones' Complement (IBFC)

Motivation: To address the limitations, especially the mis-classification, of IBF, we propose an enhanced version of IBF, namely the IBF with ones' complement (IBFC). The data structure of IBFC is the same as that of IBF, and the insertion and query operations are changed as follows.

Insertion: The insertion of IBFC is almost the same as that of IBF, except that V_e is the $(2 \cdot l)$-bit concatenation of set ID S_e and ones' complement of S_e, both in binary format. And when performing a bitwise OR operation, the length of the binary string is $2 \cdot l$.

Query: The query of IBFC is almost the same as that of IBF, except that $v_i(e)$ is a $(2 \cdot l)$-bit string starting from $B[h_i(e)]$. And after getting the bitwise AND result V_e, which is $(2 \cdot l)$-bit long, we check if the former l-bit binary string is the ones' complement of the latter l-bit binary string. And if it is the case, return the former l-bit binary string as the set ID; if the bitwise OR result of the two strings has a 0 bit, report that e does not belong to the multi-set; otherwise, return Error.

Advantages over IBF: The advantages of IBFC utilizing ones' complement are twofold. On the one hand, *ones' complement helps us to verify the correctness of an estimated set ID.* As discussed in the limitations of IBF, mis-classification may occur when using IBF. However, due to the ones' complement of every set ID, when querying the membership of an element, before returning the final result, IBFC will check if the estimated set ID match its ones' complement, which will actually eliminate the problem of mis-classification of IBF, as is proved in the next paragraph. On the other hand, *IBFC is more "fair" than IBF, regarding the elements from different sets.* In IBF, elements from different sets, when encoded into the bit strings, will write different numbers of 1, because the binary representations of different set IDs will have different numbers of 1. However, IBFC will concatenate set ID and its ones' complement, thus the number of 1 will always be l, which is half of the length of V_e. Therefore, the distribution of elements among the multi-set will not affect the performance of IBFC and elements from different sets will have the same overall accuracy. This feature helps us to determine the memory size allocated for IBFC, which is related to the number of bits to be set to 1 in B.

Error Types of IBFC: Mis-classification will never occur when using IBFC. When querying an element e, we check all bit pairs $(B[(h_i(e) + j)\%m], B[(h_i(e) + j + l)\%m])$ $(1 \leqslant i \leqslant k, 0 \leqslant j \leqslant l - 1)$, and divide them into three cases: **1)** There are one or more $(0, 0)$ bit pairs, IBFC claims that e does not belong to any of the sets. **2)** There are no $(0, 0)$ pair but one or more $(1, 1)$ pairs. Some of the 1 bits are wrong and IBFC claims that a *conflict classification* (which will be explained below) happens. For example, we get an estimated binary string $(10110\ 01101)_2$ for e whose original set ID is $(10010)_2$, and we find that the 3rd bit and 8th bit of $(10110\ 01101)_2$ are both 1, but one of the two bits is supposed to be 0, so IBFC claims that it is a conflict classification. **3)** If all pairs are $(0, 1)$ or $(1, 0)$, we use the number represented by the first half of V_e as the estimated set ID for e, and the result is correct or a false positive, but mis-classification will not happen. Because if a collision ever happened in this case, it must have changed a pair from $(0, 0)$ to $(0, 1)$ or $(1, 0)$, and mis-classification requires the element belongs to a certain set, so pair $(0, 0)$ will never exist in that case.

From the above discussion we can derive that IBFC will suffer from conflict classification and false positives but not from mis-classification [17]. *Conflict classification* means that, when querying an element e from a certain set, IBFC cannot

definitely determine the set ID of e but can provide several candidate sets one of which e must belong to.

IV. Mathematical Analysis

A. Theoretical Analysis of IBF

Let m be the number of bits in the IBF, s be the number of sets, c be the number of bits to represent a set ID, k be the number of hash functions, and n be the number of inserted elements.

Let P be the probability that the query result of any element has one more `1` bit than the correct result, which means a certain bit of the query result has been changed from `0` to `1` because of collision. Let n_i be the size of S_i, and w_i be the number of `1` bits in the binary format of set ID i, then we can calculate that:

$$P = (1 - \prod_{i=1}^{s}(1 - \frac{w_i}{m})^{n_i k})^k. \tag{1}$$

Proof. Let b be any of the `0` bit in B. During an insertion, the B will be modified by k different c-bit strings. The probability that b is covered by a c-bit string is c/m, and when it occurs, the probability that b is set to `1` is w_i/c. Therefore, the probability that b is not set to `1` by a c-bit string is $1 - w_i/m$, and the probability that b is not set to `1` after all the n_i elements from S_i have been inserted is $(1 - \frac{w_i}{m})^{n_i k}$, considering each element brings k c-bit strings. Thus, the probability that b is not set to `1` after elements from s sets have all been inserted is $\prod_{i=1}^{s}(1 - \frac{w_i}{m})^{n_i k}$. Finally, to make a query result, which is the bitwise `AND` result of k binary strings, have one more `1` bit, k corresponding bits have to be changed from `0` to `1`. Therefore, we can conclude that $P = (1 - \prod_{i=1}^{s}(1 - \frac{w_i}{m})^{n_i k})^k$. □

Let Fpr denote the probability of false positive of IBF. For IBF, the query result of an element not belonging to any of the sets is `0`, and such an element will get a correct result if none of the c bits in the bitwise `AND` result has been changed to `1`, whose probability is $(1 - P)^c$. Thus, we conclude that:

$$Fpr = 1 - (1 - P)^c. \tag{2}$$

Let P_j^{mis} be the probability of mis-classification of an element from S_j. Mis-classification occurs when several `0` bits of set ID j have been changed to `1` in the query result. There are $c - w_j$ `0` bits in set ID j, and the probability that none of them has been changed to `1` is $(1 - P)^{c - w_j}$. Therefore, we can conclude that the probability of mis-classification of an element from S_j is:

$$P_j^{mis} = 1 - (1 - P)^{c - w_j}. \tag{3}$$

According to Eq. 2 and Eq. 3, Fpr and P_j^{mis} are positively related to P. We calculate the minimum value of P in order to minimize Fpr and P_j^{mis}. According to Eq. 1, P is a function of k given s, m and n_i ($1 \leqslant i \leqslant s$). k should be set to the following value to minimize P:

$$k = -\frac{\ln 2}{\ln A}, \tag{4}$$

where $A = \prod_{i=1}^{s}(1 - \frac{w_i}{m})^{n_i}$.

Proof. Let A denote $\prod_{i=1}^{s}(1 - \frac{w_i}{m})^{n_i}$, and $P = (1 - A^k)^k$. To minimize P, we take the first-order derivative of $\ln P$ with respect to k and get the following equation:

$$\begin{aligned} \frac{\partial}{\partial k}\ln P &= \frac{\partial}{\partial k}\ln\left((1 - A^k)^k\right) \\ &= \ln(1 - A^k) - \frac{A^k \ln A^k}{1 - A^k}. \end{aligned} \tag{5}$$

Let the above formula be 0, and we get $A^k = \frac{1}{2}$, which is also a global minimum point. Therefore, Fpr and P_j^{mis} are both minimized when P is minimized with $k = -\frac{\ln 2}{\ln A}$. □

B. Theoretical Analysis of IBFC

Use the above notations and let $2 \cdot c$ be the number of bits to represent a set ID. Same as the definition of P, let P' be the probability that the query result of any element has one more `1` bit than the correct result. Then we can conclude that:

$$P' = (1 - (1 - \frac{c}{m})^{nk})^k. \tag{6}$$

Proof. Similar to the proof of Eq. 1, we replace w_i with c in the calculation of P and get

$$\begin{aligned} P' &= (1 - \prod_{i=1}^{s}(1 - \frac{c}{m})^{n_i k})^k \\ &= (1 - (1 - \frac{c}{m})^{nk})^k. \end{aligned} \tag{7}$$

□

Let Fpr' be the probability of false positive of IBFC. And we conclude that:

$$Fpr' = (1 - (1 - P')^2)^c. \tag{8}$$

Proof. IBFC reports that an element does not belong to the multi-set when the bitwise `OR` result of the former c-bit and the latter c-bit of V_e has one or more `0` bits. Therefore, false positive happens when all the c bit pairs $(V_e[j], V_e[j + c])$ $(0 \leqslant j \leqslant c - 1)$ have at least one `1` bit. The probability that $(V_e[j], V_e[j + c])$ remains `(0,0)` is $(1 - P')^2$. Therefore, the probability that all the c pairs have been changed is $(1 - (1 - P')^2)^c$, which is exactly the probability of false positive. □

The conflict classification happens when the former c-bit string of V_e is not the ones' complement of the latter c-bit string of V_e, which is because one or more `0` bits of V_e have been changed to `1`. There are initially c bits that are `0` in V_e, and the probability that none of them changed to `1` is $(1 - P')^c$. Let P_{cf} denote the probability of conflict classification, and we get:

$$P_{cf} = 1 - (1 - P')^c. \tag{9}$$

To minimize Fpr' and P_{cf}, we take the derivative of $\ln Fpr'$ with respect to P' and get the following equation:

$$\begin{aligned} \frac{\partial}{\partial P'}\ln Fpr' &= \frac{\partial}{\partial P'}(c \ln(1 - (1 - P')^2)) \\ &= \frac{2c(1 - P')}{1 - (1 - P')^2} > 0. \end{aligned} \tag{10}$$

According to the above equation, we find that Fpr' is minimized when P' is minimized. As for P_{cf}, it is minimized when P' is minimized, according to Eq. 9. Therefore, we have to find a proper value for k to minimize P'. The derivation process is similar to that of Eq. 4, and the value of k is:

$$k = -\frac{\ln 2}{\ln A} \approx \frac{m}{cn} \ln 2, \tag{11}$$

where $A = (1 - \frac{c}{m})^n \approx \mathrm{e}^{-cn/m}$. Both Fpr' and P_{cf} are minimized when $k = \frac{m}{cn} \ln 2$.

V. Performance Evaluation

A. Experimental Setup

Metrics: To evaluate the performance of IBF and IBFC, we use *false positive rate* [18] and *correct rate* to measure their accuracy, and use *number of memory accesses* [19] and *throughput* to measure their speed.

Experimental Datasets: We use 10 sets of IP traces captured by the main gateway of our campus as the experimental datasets. Each flow in those IP traces is a five-tuple: source IP address, destination IP address, source port, destination port and the protocol type. And there are 10 million packets in each set, representing approximately 1 million distinct flows and divided into 255 different sets whose ID range from `1` to `255`.

Implementation: We implement the insertion and query operations of Coded Bloom filter, Combinatorial Bloom filter, Shifting Bloom filter, IBF, and IBFC using C++ and use MurmurHash3 [20], which can hash fast and uniformly, to be their hash functions. The code length of each element is 8 bits, and in all our experiments, we allocate 6MB memory for each Bloom filter.

Computational Platform: All the experiments are performed on a machine with 12-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62GB total DRAM memory running Ubuntu 14.04.

B. Accuracy

To measure the correct rate, we first insert all the flows from one dataset into the tested Bloom filter, and then query each flow in the tested Bloom filter and check if the query result is equal to its real set ID. As shown in Fig. 2, *the correct rates of IBF and IBFC are high and comparable to that of Coded Bloom filter and Shifting Bloom filter, which are all around 95%.* Combinatorial Bloom filter shows a relatively low correct rate because it suffers from the limited memory size provided.

To measure the false positive rate, we first insert all the flows from one dataset into the tested Bloom filter, and then query flows that are not in the dataset, checking if the tested Bloom filter can correctly report that it does not belong to the multi-set. As shown in Fig. 3, *while the false positive rate of IBF is relatively high and comparable to Coded Bloom filter, it is still in an acceptable range, and the false positive rate of IBFC is comparable to that of Combinatorial Bloom filter and Shifting Bloom filter, which are all lower than 1%.*

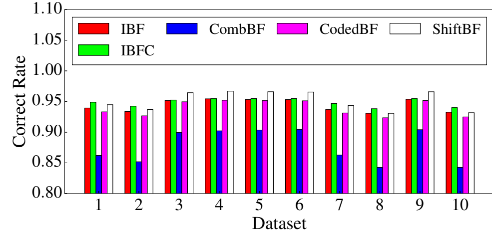

Fig. 2. Correct Rate

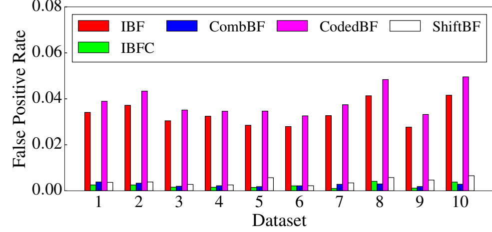

Fig. 3. False Positive Rate

From the experimental results on correct rate and false positive rate, we can conclude that the performances of IBF and IBFC are comparable to prior art in terms of accuracy.

C. Speed

In this section, we measure the speed of IBF and IBFC in terms of number of memory accesses per insertion and query, and query throughput. We set the number of hash functions to 3 in the following experiments.

As is shown in Fig. 4, *number of memory accesses per insertion of IBF and IBFC, which are 4 and 8 respectively, are observably lower than that of Combinatorial Bloom filter, Coded Bloom filter, and Shifting Bloom filter, which can be as high as 20,* which proves that the speed of IBF and IBFC is indeed higher than the prior art in terms of number of memory accesses per insertion.

As is shown in Fig. 5, *number of memory accesses per query of IBF and IBFC, which are 4 and 10 respectively, are observably lower than that of Combinatorial Bloom filter, Coded Bloom filter, and Shifting Bloom filter, which can be as high as 30,* which proves that the speed of IBF and IBFC is indeed higher than the prior art in terms of number of memory accesses per query.

As discussed in section III, IBF and IBFC directly record the set ID into the Bloom filter, instead of combining different Bloom filters. Therefore, IBF and IBFC are expected to have a smaller number of memory accesses. More specifically, suppose each machine word is 64 bits and can be retrieved by one memory access. To query one element with set ID 255, 1 or 2 machine words are accessed using IBF or IBFC, whereas 4 or 5 machine words are accessed using Shifting Bloom filter.

As shown in Fig. 6, *the throughput of IBF and IBFC, which are 0.4 Mips and 0.3 Mips respectively, are observably higher than that of Combinatorial Bloom filter, Coded Bloom filter,*

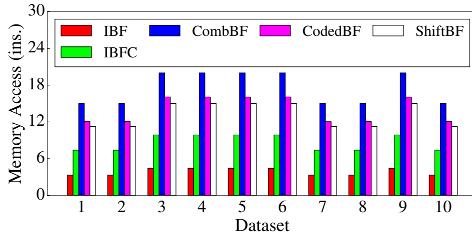

Fig. 4. Memory Access per Insertion

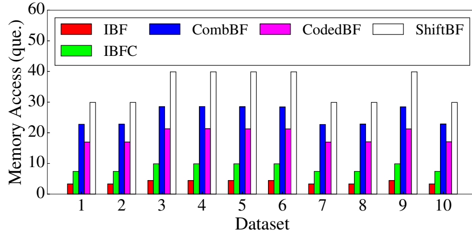

Fig. 5. Memory Access per Query

and Shifting Bloom filter, which is reasonable, since IBF and IBFC have a smaller number of memory accesses.

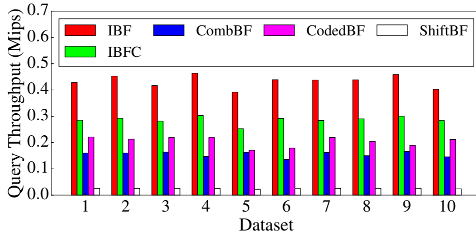

Fig. 6. Throughput of Query

From the above experimental results on accuracy and speed of IBF and IBFC, we can conclude that their accuracy are high enough, and their speed are much higher compared to the prior art. IBF and IBFC can be applied to many practical network applications.

VI. Conclusion

Multi-set membership query is a fundamental operation in many network applications such as firewalls and switches. In this paper, we propose a novel data structure, namely ID Bloom filter (IBF), which only needs a small constant number of memory accesses for each query while maintaining high accuracy by directly recording the set IDs in a bit array. To further improve accuracy, an enhanced version, IBF with ones' complement (IBFC), is proposed. Extensive experimental results show that IBF and IBFC achieve a much higher speed than the state-of-the-art. IBF and IBFC can be well applied to many real-world network applications. All the related source code has been released on GitHub [21].

References

[1] Myung Keun Yoon, JinWoo Son, and Seon-Ho Shin. Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In *INFOCOM, 2014 Proceedings IEEE*, pages 1429–1437. IEEE, 2014.

[2] Francis Chang, Kang Li, and Wu-chang Feng. Approximate packet classification caching. In *Proc. IEEE INFOCOM,*, 2003.

[3] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[4] Shahabeddin Geravand and Mahmood Ahmadi. *Survey Bloom filter applications in network security: A state-of-the-art survey*. Elsevier North-Holland, Inc., 2013.

[5] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *Proc. ACM SIGCOMM*, volume 44, pages 39–50, 2014.

[6] Tong Yang, Bo Yuan, Shenjiang Zhang, Ting Zhang, Ruian Duan, Yi Wang, and Bin Liu. Approaching optimal compression with fast update for large scale routing tables. In *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service*, page 32. IEEE Press, 2012.

[7] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Bloom filters: Design innovations and novel applications. In *43rd Annual Allerton Conference*, 2005.

[9] Fang Hao, Murali Kodialam, T. V. Lakshman, and Haoyu Song. Fast dynamic multiple-set membership testing using combinatorial bloom filters. *IEEE/ACM Transactions on Networking*, 20(1):295–304, 2012.

[10] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[11] Fang Hao, Murali Kodialam, T V Lakshman, and Haoyu Song. Fast dynamic multiset membership testing using combinatorial bloom filters. *Proceedings - IEEE INFOCOM*, 20(1):513–521, 2009.

[12] Sailesh Kumar and Patrick Crowley. Segmented hash:an efficient hash table implementation for high performance networking subsystems. In *ACM/IEEE Symposium on Architecture for NETWORKING and Communications Systems, ANCS 2005, Princeton, New Jersey, Usa, October*, pages 91–103, 2005.

[13] F. Chang, Wu Chang Feng, and Kang Li. Approximate caches for packet classification. In *Joint Conference of the IEEE Computer and Communications Societies*, pages 2196–2207 vol.4, 2004.

[14] Bernard Chazelle, Joe Kilian, Ayellet Tal, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Fifteenth Acm-Siam Symposium on Discrete Algorithms*, pages 30–39, 2004.

[15] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.

[16] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.

[17] YuXin Liu, Anfeng Liu, Shuang Guo, Zhetao Li, Young-June Choi, and Hiroo Sekiya. Context-aware collect data with energy efficient in cyber–physical cloud systems. *Future Generation Computer Systems*, 2017.

[18] Farzaneh Sadat Tabataba and Masoud Reza Hashemi. Improving false positive in bloom filter. In *Electrical Engineering (ICEE), 2011 19th Iranian Conference on*, pages 1–5. IEEE, 2011.

[19] Jun Xu, Mukesh Singhal, and Joanne Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1445–1454. IEEE, 2000.

[20] Source code of the murmurhash3. https://chromium.googlesource.com/external/smhasher/+/c2b49e0d2168979b648edcc449a36292449dd5f5/MurmurHash3.cpp.

[21] Source code of the *IBF/IBFC* and related data structures. https://github.com/BlockLiu/IDFilter.