Elastic Bloom Filter: Deletable and Expandable Filter Using Elastic Fingerprints

Yuhan Wu*, Jintao He*, Shen Yan*, Jianyu Wu*, Tong Yang*†, Olivier Ruas‡, Gong Zhang§, Bin Cui*
*Department of Computer Science and Technology, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
†Peng Cheng Laboratory, Shenzhen, China ‡Inria, Univ. Lille §Huawei Theory Lab, China

Abstract—The Bloom filter, answering whether an item is in a set, has achieved great success in various fields, including networking, databases, and bioinformatics. However, the Bloom filter has two main shortcomings: no support of item deletion and no support of expansion. Existing solutions either support deletion at the cost of using additional memory, or support expansion at the cost of increasing the false positive rate and decreasing the query speed. Unlike existing solutions, we propose the Elastic Bloom filter (EBF) to address the two shortcomings simultaneously. Importantly, when EBF expands, the false positives decrease. Our key technique is **Elastic Fingerprints**, which dynamically absorb and release bits during compression and expansion. To support deletion, EBF can first delete the corresponding fingerprint and then update the corresponding bit in the Bloom filter. To support expansion, Elastic Fingerprints release bits and insert them to the Bloom filter. Our experimental results show that the Elastic Bloom filter significantly outperforms existing works.

Index Terms—Bloom filter, Elastic Bloom filter, Elastic Fingerprints, Bloom filter expansion

1 INTRODUCTION

The Bloom filter [7], a highly compact probabilistic representation of a set, is used to answer whether a particular item is in the set. A standard Bloom filter is a bit array along with k hash functions. The hash functions are used to map each item into k positions/bits in the array, and we call them the k *mapped bits*. An element is inserted by setting its k mapped bits to 1 while querying the presence of an element is done by checking if all the k mapped bits are set to 1. The main advantages of Bloom filters are: *(i)* small memory footprint, *(ii)* fast and constant speed of queries and updates, *(iii)* no false negatives, small and tunable false positive rate. Due to these advantages, the Bloom filter and its variants have been widely used in a great many fields, such as real-time systems [24], computer architectures [21], neural network [17], IP lookups [10], [18], [23], web caching [13], Internet measurement [11], packet classification [38], regular expression matching [9] , multicast [32], queue management [8], routing [31], [35], P2P networks [20], [30], data center networks [39], cloud computing [26], and more [16], [28], [37].

However, the Bloom filter suffers from two main drawbacks: they are not deletable nor expandable. More specifically, 1) once an item has been inserted into the standard Bloom filter, the item cannot be directly deleted; 2) once a set has been inserted into the Bloom filter, it is impossible to construct a larger Bloom filter representing the same set without extra information. In many applications, the Bloom filter is the best choice. However, these applications inevitably suffer from the aforementioned drawbacks as discussed in the following examples:

- Black lists. The Bloom filter is used to store a black list to prevent threats such as DDoS attack [27] and amplification attack [33]. However, when an IP address is queried, even if the IP address is legal, the result might be a false positive and the IP address is regarded it as malicious. To solve the problem, a white list can be set up to store those friendly addresses. In practice, the elements of both lists might change and the lists should be deletable and adjustable.
- MAC address lookup. Each switch has a MAC address table. The table has a large number of entries and each entry can be considered as a Key-Value pair. The key is the destination MAC address and the value is the outgoing port. One Bloom filter is built for all MAC addresses with the same outgoing port [39]. The size of MAC address table dynamically changes, and could increase a lot.
- Multicast routing. Multicast routing is the routing of IP multicast datagrams. The Bloom filter is used to compress a Multicast forwarding table for each outgoing interface in the switch [29]. The Bloom filter in each outgoing interface is used to determine whether to forward an incoming packet or not. The member of an interface will join and leave the forwarding tables. The size of forwarding table changes dynamically and is unknown in advance, and the size of a forwarding table could be quite large.
- Longest prefix matching (LPM). The LPM is part of the rule of Internet Protocol (IP) routing. The LPM is to find an address with the largest number of same leading bits. For LPM, the Bloom filter is used [10] to determine the length of the matching prefix. The candidates of the LPM, the entries containing addresses, change dynamically and the number of candidates can be enormous.

Co-primary authors: Yuhan Wu, Jintao He, and Shen Yan. Corresponding authors: Tong Yang (yangtongemail@gmail.com). E-mail: {yuhan.wu, 16hjt, yanshen, jywu2017, bin.cui}@pku.edu.cn, yangtongemail@gmail.com, olivier.ruas@inria.fr, nicholas.zhang@huawei.com

The above examples require the Bloom filter to support item deletion and expansion. In summary, when Bloom filters are used for representing dynamic sets, deletion and expansion are often indispensable.

Some existing works focus on addressing one of the above two shortcomings of Bloom filters, but none of them can address both shortcomings at the same time without sacrificing the query efficiency. To support item deletions, the counting Bloom filter (CBF) [13] stores counters instead of bits in the Bloom filter. A common approach is to maintain a counting Bloom filter in slow memory to support the deletion of members and a Bloom filter in fast memory to support fast queries. But the second shortcoming of expansion cannot be addressed by using CBF. To support expansion, Scalable Bloom filters [36] and Dynamic Bloom filters [14], [34] append a new empty Bloom filter at the end of the the old structure repeatedly. The optimized Dynamic Bloom filter [15] replaces each Bloom filter with a counting Bloom filter to support item deletion. The overhead of these solutions is that the query time and the false positive rate are persistently increasing along with the number of new Bloom filters. In contrast, the design goal of this paper is to address the two shortcomings of the Bloom filter simultaneously with neither additional query overhead nor additional accuracy loss.

In this paper, we propose a novel data structure, namely the *Elastic Bloom filter* (EBF), that overcomes the above two shortcomings at the same time. EBF is an extension of the standard Bloom filter that supports both *item deletion* and *expansion*. Contrary to the Dynamic Bloom filters whose expansion will increase the false positive rate, the expansion of the EBF can significantly reduce its false positive rate.

The key technique of the EBF is called ***Elastic Fingerprints***. EBF consists of a standard Bloom filter and an elastic fingerprint array. To expand the Bloom filter, we first cut one bit from each fingerprint, and appropriately combine the Bloom filter and the cut bits into a larger Bloom filter. To compress the Bloom filter, some bits of the Bloom filter are to be lost, while we append these lost bits to the elastic fingerprints. In other words, EBF dynamically moves bits between the fingerprint array and the Bloom filter during expansion and compression. For deletion, we first delete fingerprints from the fingerprint array, and then determine which bits in the Bloom filter should be cleared. Further, we propose two optimization methods namely *lazy update* and *bucket tree* in Section 3.3. We have open-sourced all code at GitHub [2].

2 BACKGROUND AND RELATED WORK

2.1 Standard Bloom filter

The Bloom filter is a highly compact probabilistic representation of a set and it answers whether a queried item is in the set. Let $U = \{e_1, .., e_n\}$ be the universal set of items, and $S \subset U$ a subset of U. A Bloom filter of S is to test whether a given item $e \in U$ belongs to S. In practice, the set S is the result of successive insertions and deletions of items. In that context, an item e is in S if it has been inserted in S and has not been removed from it.

A Bloom filter is a bit array A of size m along with k hash functions $(h_i)_{i=1,..,k}$. The hash functions map the items of U to bits of A: $\forall i \in [1,k], \forall e \in U, h_i(e) \in [0,m)$. An empty Bloom filter has all its bits set to 0.

Insertion: To insert an item e to the filter, we compute all its hashes by the hash functions and set the mapped bits to 1: $\forall i \in [1,k], A[h_i(e)] \leftarrow 1$

Query: To check whether an item e has been inserted into S, we check whether all its mapped bits in A are 1: $\wedge_{i=1}^{k} h_i(e) == 1$. The queries do not cause false negatives but some false positives may occur. For a Bloom filter of size m with k hash functions and in which n different items have been inserted, the false positive rate (FPR) [7] is $\delta = (1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k$. Note that the standard Bloom filter does not support item deletion and its size never changes.

2.2 Related Work

The Bloom filter [7], [22] is widely used because of its three main advantages: 1) memory efficient; 2) fast to query and update; and 3) no false negatives. Many variants of the Bloom filter have been designed to improve its performance. In this paper, we only focus on the variants about its expansion and deletionand analyze them from the aspect of the above three main advantages. In order to support item deletion, the Counting Bloom filter (CBF) [13] replacing each bit by a counter. When inserting an item, its mapped counters are increased. Similarly, its mapped counters are decreased when removing an item. This additional feature comes at an important memory overhead. A. Pagh, R. Pagh & S. S. Rao [25] and B. Fan et al. [12] also implemented the item deletion at a lower cost. Unfortunately, those approaches do not allow expansion, the number of inserted items must be known in advance.

In order to support the dynamic set well, Scalable Bloom filters [36] and Dynamic Bloom filters (DBF) [14], [34] provide an adaptable size to the Bloom filter. Once a Bloom filter is considered to be full, i.e. its estimated false positive rate is higher than a given threshold, a new Bloom filter is appended. While DBF adds similar Bloom filters, Scalable Bloom filters append filters with larger sizes to have better control over the false positive rate. Unfortunately, it results in higher response time for queries, as the items should be queried successively in several Bloom filters. Also, DBF and Scalable Bloom filters suffer from the same shortcoming as the standard Bloom filter: they do not support item deletion.

The optimized Dynamic Bloom filter [15] and its variant Par-BF [19] replace each Bloom filter of DBF with a CBF to support item deletion. Therefore, they can support both item deletion and expansion. However, they still suffer from the linearly increasing query time and false positives, which is hardly acceptable for query-oriented scenarios, including the four scenarios introduced in Section 1.

3 ELASTIC BLOOM FILTER

3.1 Structure

The Elastic Bloom filter consists of two parts, a standard Bloom filter, and a cooperative bucket array. The Bloom filter is stored in the fast memory to provide fast queries while the cooperative bucket array, used for expansion, is stored in the slow memory. The standard Bloom filter is an array with m bits. And the cooperative bucket array is an array containing

m buckets. Every bit in the Bloom filter is associated to the bucket with the same index in the bucket array. We use k independent hash functions. Each hash function hashes the item into a w-bit hash number. We use uniform random hash functions: the output numbers are uniformly distributed in the range $[0, 2^w)$. The hash number is separated into two distinct parts by dividing it by m: the quotient and the remainder. *The index* of an item in the Bloom filter is the remainder while the quotient, named *the Elastic Fingerprint*, is stored in the cooperative bucket associated with the index. Each bucket can store D Elastic Fingerprints.

Notations: Let A denote the bit array and B denote the cooperative bucket array. They both have a size m. The k hash functions $(h_i)_{i\in[1,k]}$ hash item e into a pair $(F_p,\ I_{ndex})_i$ where F_p is the fingerprint in the range $\left[0, \left\lfloor \frac{2^w}{m} \right\rfloor\right)$ and I_{ndex} is the index in the range $[0, m)$.

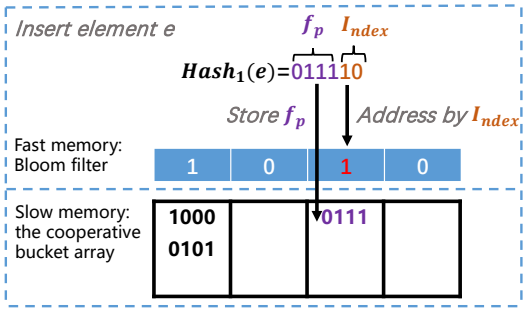


Fig. 1: Insertion of the Elastic Bloom filter. The Bloom filter size $m = 4$, the length of hash functions $w = 6$ and number of hash functions $k = 1$. When inserting an item e, we first calculate its binary hash number 011110_2 by h_1. Next, we divide the hash number by m to get the $F_p = 0111_2$ and $I_{ndex} = 10_2$. Then, we set $A[10_2]$ to 1 in the Bloom filter and insert the fingerprint 0111_2 into bucket $B[10_2]$.

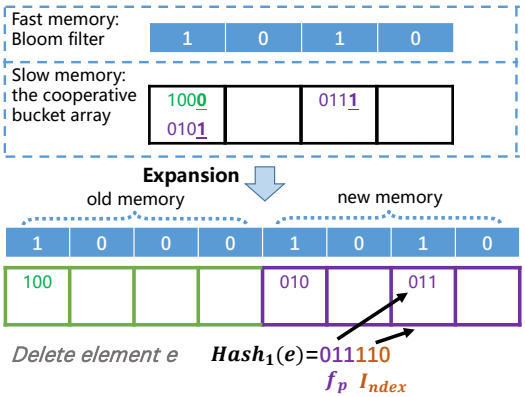


Fig. 2: Expansion and deletion of the Elastic Bloom filter. The size of the Bloom filter $m = 4$. We first allocate new memory for the new structure and append the new structure to the end of the old structure. We scan all the old buckets to reallocate all the fingerprints by their lowest bits. The lowest bit of each fingerprint indicates whether it should be moved to the new part (purple color) or keep its location (green color). For example, 0101_2 in bucket $B[0]$ is a fingerprint that needs to be moved, it will be moved to bucket $B[0 + m] = B[4]$. Then we update the Bloom filter by the bucket array. Set $A[2]$ to 0 because the bucket $B[2]$ is empty. And set $A[4]$, $A[6]$ to 1 because the bucket $B[4], B[6]$ have fingerprints.
If we want to delete the item e which is inserted before, we should calculate the $F_p = 011_2$ and $I_{ndex} = 110_2$ based on the new Bloom filter size in the same way as insertion. Then, we delete the F_p from $B[I_{ndex}]$ and reset the corresponding bit in Bloom filter to 0 if the bucket is empty.

3.2 Operations

We now introduce the operations of the EBF: item insertion, item query, item deletion, the expansion of the EBF, and the compression.

3.2.1 *Insertion:*

To insert an item e into the EBF, we first compute the hash numbers using the k hash functions. We obtain k pairs $(F_p,\ I_{ndex})_i$. We set all the mapped bits of A to 1 and insert the fingerprints to the associated buckets in B:

$$\forall i \in [1,k], A[h_i(e).I_{ndex}] \leftarrow 1$$
$$\forall i \in [1,k], B[h_i(e).I_{ndex}] \leftarrow h_i(e).F_p$$

The updates of A and B can be done independently. Fig. 1 shows the insertion of an item e into the EBF. If the insertion operation insert an item that has been inserted, we should check whether it has been inserted in order to avoid duplicate fingerprints. Before we set all the mapped bits of a to 1, we check whether there is any mapped bit that is 0. If so, we directly insert the fingerprints into the bucket, because it must have not been inserted. Otherwise, we additionally check whether there is the same fingerprint in every associated bucket. If so, we believe that the item is a duplicate and we do not insert its fingerprints. Otherwise, we just insert the fingerprints to the buckets.

3.2.2 *Query:*

To query whether an item e is in the EBF, we compute all the hash indexes $h_i(e).I_{ndex}$ and return *true* if all the mapped bits in A are set to 1, *false* otherwise. The result is $\wedge_{i=1}^{k} A[h_i(e).I_{ndex}]$. There is no need to access the slow memory so that we can achieve a high query speed.

3.2.3 *Deletion:*

To remove an item, we compute its hash indexes and fingerprints, and then delete its fingerprints from the mapped buckets. If a bucket becomes empty due to the deletion, its corresponding bit in the Bloom filter is set to 0. Note that only the deletion of a previously inserted item is allowed.

3.2.4 *Expansion:*

The Elastic Bloom filter expands automatically **(Case 1)** when inserting a fingerprint into a full bucket or **(Case 2)** when the ratio of 1 bits in the Bloom filter, namely Set Bit Rate (SBR), is above a threshold Ω. In other words, the False Positive Rate (FPR) is above a threshold $\Delta = \Omega^k$.

To expand our EBF, we double the allocated memory space of both the Bloom filter in the fast memory and the cooperative bucket array in the slow memory. The new memory is appended to the existing EBF. All the new bits are initiated to 0 and all the new buckets are empty. We now spread the fingerprints in the old EBF around the two parts of the new EBF. To do so, we scan the m buckets one by one to reallocate all the fingerprints. For every pair $(F_p,\ I_{ndex})$, the hash number of it is $F_p * m + I_{ndex}$.

Now that the size is $2m$ after expansion, the new fingerprint should be the quotient by $2m$, and the index should be the new remainder. In other words, the lowest bit of F_p determines its new position: if it is 0, the item should remain in the same bucket $B[I_{ndex}]$, otherwise it should be moved to bucket $B[I_{ndex} + m]$. In both cases, the value of the fingerprint is divided by two: $F_p \leftarrow \lfloor F_p/2 \rfloor$. After

reallocating all fingerprints, we update the Bloom filter: for a given index $i \in [0, 2m)$, if $B[i]$ is empty, we set $A[i]$ to 0. Otherwise, we set $A[i]$ to 1. Fig. 2 shows the expansion.

3.2.5 Compression:

The compression is the inverse process of expansion. Suppose that the EBF has m buckets, where m is an even number. For each bucket $B[i], i = 0, 1, \ldots, m/2$, we merge it with bucket $B[i + m/2]$. When merging the two buckets, we append one bit to each fingerprint and store the new fingerprint in the merged bucket. For the fingerprints from bucket $B[i]$, we append a zero bit (multiply the fingerprint by 2). For the fingerprints from bucket $B[i + m/2]$, we append a bit of 1 (multiply the fingerprint by 2 and add one). The EBF compresses automatically when the estimated Set Bit Rate (SBR) is below a threshold $\frac{1}{4}\Omega$ and it should not cause bucket overflow.

3.3 Optimizations

3.3.1 Lazy update:

In network applications, the expanding operation should be done as fast as possible to minimize the loss of packets. The lazy update is an optimization to make the expanding operation much faster. When expanding, the first thing we do is to copy the old Bloom filter into the newly allocated memory. In this way the query can be performed right away.

Then, instead of spreading the fingerprints around the old and new memory, we simply copy the old bucket array into the new one. Besides, we attach a sign bit to every bucket. The sign bit, which is initialized to 0, indicates whether the bucket and its corresponding bit in the Bloom filter have been updated. If a bucket has been updated (e.g., reallocate all its fingerprints and renew the corresponding bit in Bloom filter), the sign bit will be set to 1. We update the buckets through two mechanisms in parallel: *(i)* the scanning function that scans each bucket and updates them and *(ii)* the updating with insertion function that updates buckets whenever the buckets are accessed to add or remove an item. When all the buckets have been updated, the structure can support expansion once again.

3.3.2 Bucket trees:

To prevent the buckets from overflowing, we reorganize the structure of the cooperative bucket array. The new structure is a bucket tree: a two-level k-ary tree whose nodes are buckets. Every bucket can store up to S fingerprints. There are m buckets in level 1. The i-th bucket in level 1 is associated to the $\lfloor \frac{i}{k} \rfloor$-th bucket in level 2. When inserting a fingerprint, we first attempt to insert it into the bucket in level 1. If the bucket in level 1 is full, we insert it to the associated bucket in level 2 and add $\lceil log_2(k) \rceil$ bits at the beginning of the fingerprint to indicate where the fingerprint is inserted from. If the bucket in level 2 still overflows, we suspend the insert operation and expand our structure instantly. When expanding the structure, we try to pull the fingerprints in level 2 back to level 1. After expansion, we try to insert the item again.

3.4 Extra Operations

The main purpose of the Elastic Bloom filter is to provide deletion and expansion to the Bloom filter while still providing both fast answers and a low false positive rate. Additionally, EBF provides other operations without extra overhead: *(i)* union, *(ii)* intersection, *(iii)* cardinality, and *(iv)* accurate query.

Union of Elastic Bloom filters: Let two Elastic Bloom filters ebf_1 and ebf_2 have the same initial size and hash functions. The union of ebf_1 and ebf_2 is calculated by first expanding both filters to the maximum of their sizes. Then, their buckets $(B_i)_{i\in[1,m]}$ are merged by keeping all the fingerprints. The bit of the standard Bloom filter A is set to 1 when its corresponding bucket is not empty.

Intersection of Elastic Bloom filters: The intersection between two Elastic Bloom filters is calculated similarly as the union: the only difference is that we take the intersection of the fingerprints while merging two buckets. Note that a fingerprint may be several times in the buckets.

Cardinality: We can know exactly *how many* items have been inserted by counting the fingerprints. The insertion of every item adds k fingerprints in total, so the total number of inserted items is $\frac{1}{k}\sum_{i=0}^{m-1}|B[i]|$.

Accurate Query: The Elastic Bloom filter can provide a more accurate estimation at the expense of slower speed. The accurate query first performs the standard query: given an item e, we verify all the bits $A[h_i(e).I_{ndex}]$. If they are all set to 1, we check the presence of the fingerprints in all corresponding buckets:

$$\bigwedge_{i=1}^{k} h_i(e).F_p \in B[h_i(e).I_{ndex}]$$

4 MATHEMATICAL ANALYSIS

In this section, we provide a mathematical analysis of the Elastic Bloom filter (EBF). First, in section 4.1, we focus on the part of EBF in the fast memory, i.e., a standard Bloom filter. Then, in section 4.2, we analyze the part of EBF in the slow memory, i.e., a bucket array. Finally, We summarize how to configure the parameters of EBF in Section 4.3.

4.1 The Performance of EBF in the Fast Memory

We show the properties of the fast memory part of EBF in this section, including accuracy, space complexity, and time complexity. The data structure of EBF in the fast memory is exactly a standard Bloom filter and shares its properties. For an EBF of size m with k hash functions in which n different items have been inserted, the **false positive rate** is $\delta = [1 - (1 - \frac{1}{m})^{kn}]^k \approx (1 - e^{\frac{-kn}{m}})^k$ [7]. Conversely, when we want to achieve a given false positive rate δ while inserting n elements, the size of the filter should be at least $m = \log_2(e) n \log_2\left(\frac{1}{\delta}\right)$, and the optimal number of hash functions is $k^* = \ln(2)\frac{m}{n} = O(\ln\left(\frac{1}{\delta}\right))$. Therefore, the **space complexity** of the fast memory part is $O(n \ln\left(\frac{1}{\delta}\right))$. The **time complexity** of insertion/query/deletion is $O(\ln\left(\frac{1}{\delta}\right))$.

4.2 The Performance of EBF in the Slow Memory

We show the properties of the slow memory part of EBF (i.e., the cooperative bucket array), including space complexity and time complexity. For space complexity, we show that a small bucket size, for example $O\left(\frac{\ln(n\ln\left(\frac{1}{\delta}\right))}{\ln\ln(n\ln\left(\frac{1}{\delta}\right))}\right)$, is sufficient enough to ensure a high probability of not triggering

expansion. For time complexity, we show that applying expansion and compression operations does not increase the average time complexity, which is still $O(\ln(\frac{1}{\delta}))$ under our EBF algorithm.

First, we prove the following inequality:

Lemma 4.1. *$\forall b \in (0, \frac{1}{9})$, $a > 4$, and $M \geq \frac{a\ln(b^{-0.5})}{\ln\ln(b^{-0.5})}$, we have*

$$\left(\frac{a}{M}\right)^M \leq b. \tag{1}$$

Proof.

$$\begin{aligned}\left(\frac{a}{M}\right)^M &\leq \left(\frac{a\ln\ln(b^{-0.5})}{a\ln(b^{-0.5})}\right)^{a\ln(b^{-0.5})/\ln\ln(b^{-0.5})} \\ &= e^{-a\ln(b^{-0.5})*\left(1-\frac{\ln\ln\ln(b^{-0.5})}{\ln\ln(b^{-0.5})}\right)} \\ &= b^{0.5a*\left(1-\frac{\ln\ln\ln(b^{-0.5})}{\ln\ln(b^{-0.5})}\right)} \leq b^{0.25a} \leq b\end{aligned}$$

□

Applying Lemma 4.1, we have the following theorem.

Theorem 4.1. *For an EBF of size m with k hash functions, the space complexity of the bucket array should be at least $O\left(\frac{kn(\ln(m)-\ln(\epsilon))}{\ln(\ln(m)-\ln(\epsilon))}\right)$ to ensure the probability of not triggering expansion is at least $1-\epsilon$ when there are n distinct items in the EBF. We set $k = \Theta(\ln(\frac{1}{\delta}))$, $m = \Theta(kn)$, and $\epsilon = \frac{1}{m}$, then the space complexity is $O\left(n\ln(\frac{1}{\delta}) \cdot \frac{\ln(n\ln(\frac{1}{\delta}))}{\ln\ln(n\ln(\frac{1}{\delta}))}\right)$.*

Proof. The probability that bucket 1 receives at least M fingerprints is at most

$$\binom{kn}{M}\left(\frac{1}{m}\right)^M \leq \frac{\left(\frac{nk}{m}\right)^M}{M!} \leq \left(\frac{\frac{enk}{m}}{M}\right)^M. \tag{2}$$

Applying the union bound, the probability that one of the buckets in the bucket array receives at least M fingerprints is $m\binom{kn}{M}\left(\frac{1}{m}\right)^M$. Applying Lemma 4.1, where $a = \frac{enk}{m}$, $b = \frac{\epsilon}{m}$, we conclude that, for $M \geq \frac{enk}{m} \cdot \frac{\ln((\epsilon/m)^{-0.5})}{\ln\ln((\epsilon/m)^{-0.5})}$,

$$m\binom{kn}{M}\left(\frac{1}{m}\right)^M \leq m\left(\frac{\frac{enk}{m}}{M}\right)^M \leq \epsilon. \tag{3}$$

□

Theorem 4.2. *For any n insertion or deletion operations, the time complexity is $O(n\ln(\frac{1}{\delta}))$.*

Proof. For each insertion or deletion operation, without considering expansion or compression, the time complexity is $O\left(\ln\left(\frac{1}{\delta}\right)\right)$. So we only consider expansion/compression in the following parts. For each expansion/compression, we scan all buckets and move the fingerprints, so the time complexity is $O(m+nk) = O(n\ln(\frac{1}{\delta}))$. To find out the total time complexity, we will compute the time complexity of expansion/compression caused by the change of SBR (Case 1), and the time complexity of expansion caused by bucket overflow (Case 2), respectively.

For Case 1, suppose that the $i^{th}, i = 1, 2, \ldots, t$ expansion/compression (all triggered by Case 1) happens at the a_i^{th} insertion/deletion, we will prove the time complexity of the i^{th} expansion/compression is $O\left(\ln\left(\frac{1}{\delta}\right)(a_i^{th} - a_{i-1}^{th})\right)$, and therefore the total complexity is $O\left(\sum_{i=1}^{t}\left[\ln\left(\frac{1}{\delta}\right)(a_i^{th} - a_{i-1}^{th})\right]\right) = O(n\ln\left(\frac{1}{\delta}\right))$. For $i = 1$, $a_{i-1}^{th} = 0$ and the time complexity meets the requirement. We have two equations $\omega_{i-1} = 1 - e^{-\frac{n_{i-1}k}{m_{i-1}}}$ and $\omega_i = 1 - e^{-\frac{pn_{i-1}k}{m_i}}$. We can derive that $p = \frac{m_i\ln(1-\omega_1)}{m_{i-1}\ln(1-\omega_0)}$, which is a constant. As $m_i = 2m_{i-1}$ or $m_i = \frac{1}{2}m_{i-1}$, we have $p \in (\frac{1}{2}, \frac{5}{8}) \cup (\frac{8}{5}, 2)$. Then, we have $(1-p)n_i + pn_i = pn_{i-1}$, and therefore $n_i = \frac{p(n_{i-1}-n_i)}{1-p} \leq \left|\frac{p}{1-p}\right|(a_i^{th} - a_{i-1}^{th}) \leq \frac{8}{3}(a_i^{th} - a_{i-1}^{th})$. As the time complexity of expansion/compression is $O(n\ln\left(\frac{1}{\delta}\right))$, we derive that the time complexity of the i^{th} expansion/compression is $O\left(\ln\left(\frac{1}{\delta}\right)(a_i^{th} - a_{i-1}^{th})\right)$ and the total time complexity of Case 1 is $O(n\ln\left(\frac{1}{\delta}\right))$.

For Case 2, similarly, we will prove that the time complexity of the expansion (Case 2) happens in range $a_{i-1}^{th} \sim a_i^{th}$ (where the i^{th} expansion/compression triggered by Case 1 happens at the a_i^{th} insertion/deletion) is $O\left(\ln\left(\frac{1}{\delta}\right)(a_i^{th} - a_{i-1}^{th})\right)$, and therefore the total complexity is $O(n\ln\left(\frac{1}{\delta}\right))$. For each insertion in range $a_{i-1}^{th} \sim a_i^{th}$, according to Theorem 4.1, the probability that the insertion triggers bucket overflow is smaller than $\epsilon = \frac{1}{m}$. Applying the union bound, we get the time complexity of expansion (Case2) is smaller than $\sum_{j=1}^{\infty}\left[\frac{\epsilon}{2^j}(a_i^{th} - a_{i-1}^{th}) \cdot O(n_i\ln\left(\frac{1}{\delta}\right))\right] = O\left(\ln\left(\frac{1}{\delta}\right)(a_i^{th} - a_{i-1}^{th})\right)$, where j denotes the number of consecutive triggers of expansion (Case 2).

□

4.3 EBF Parameter Configuration

We introduce how to configure the parameters of EBF. First, we configure the EBF size m and the number of hash functions k. If you want to achieve best accuracy with least space, for the common set size n and the required false positive rate δ, you can set $m = \log_2(e)n\log_2\left(\frac{1}{\delta}\right)$ and $k = -\log_2(\frac{1}{\delta})$. But, in applications, we often choose a smaller k (e.g., $k = 4$) to achieve a better processing speed. Then m should be at least $m = -\frac{kn}{\ln(1-\delta^{\frac{1}{k}})}$ and Next, we configure the expansion threshold $\Omega = \delta^{\frac{1}{k}}$. We set $\epsilon = \frac{1}{m}$ and the bucket size $D = \frac{enk}{m} \cdot \frac{\ln((\epsilon/m)^{-0.5})}{\ln\ln((\epsilon/m)^{-0.5})}$. For hash value length w, we set w to 32, because the space of the slow memory (e.g., disk) is sufficient and the size of the Bloom filter in the fast memory can be up to $512MB$, which is larger enough for many applications.

5 EXPERIMENT RESULT

5.1 Experiment Setup

5.1.1 Datasets

The following datasets are used in our experiments.

- **CAIDA:** As many papers [4], [5] do, we use anonymized IP trace streams from CAIDA [3], and then identify each flow of IP trace streams by the five-tuples.
- **Distinct Stream:** we use Distinct Stream, which consists of random distinct elements, to test the worst case performance of each algotirhm.
- **IMC Data Center IP Trace:** We use IP trace streams collected by [6] to measure the performance of different kinds of Bloom filters in data centers. Each flow of IP trace streams is identified by the five-tuples.

5.1.2 Implementation

We implement our algorithm in C++. The programs are run on a server with 18-core CPU (36 threads, Intel CPU i9-10980XE @3.00 GHz) and 128GB memory. In all experiments, we use MurmurHash [1], a well-acknowledged hash function, to calculate the hashing value of elements. We compare our EBF with the Dynamic Bloom filter (**DBF**) [14], partitioned BF (**Par-BF**) [19], the Scalable Bloom filter (**SBF**) [36], and the Counting Bloom filter (**CBF**) [13]. The default parameters of our experiment are listed in TABLE 1.

Hashnum	Bucket size	BF size (2^k)	W/WO Bucket tree
5	8	15	W

Branches	Dataset	Expanding threshold
3	Distinct Stream	0.2

TABLE 1: default experiment parameters

5.1.3 Evaluation Metrics and Key Parameters

- **FPR (False Positive Rate):** $\delta = \frac{n}{m}$, where m denotes the number of queried items that do not appear in the Bloom filter and n denotes the number of items that are mistaken as the items in the Bloom filter. We use FPR to evaluate the accuracy of a Bloom filter.
- **SBR (Set Bit Rate):** $\omega = \frac{L}{m}$, where m denotes the size of our Bloom filter and L denotes the number of bits that are set to 1 in the Bloom filter.
- **Throughput:** We use MOPS (Million Operations Per Second) to measure the speed of each algorithm.
- **LR (Load Rate):** $\frac{I}{T}$, where T denotes the total number of fingerprint slots in the buckets and I denotes the total number of fingerprints that can be inserted into the Bloom filter without overflow. We use LR to evaluate the loading ability of our buckets in the slow memory.
- **RPDS (Relative Peak Data Size):** $\frac{n_1}{n_0}$. Typically, the number of elements in the Bloom filter is smaller than a predetermined threshold n_0, which does not trigger expansion. In special cases, the number of elements changes dynamically, reaching a peak value n_1. We call the ratio of n_1 to n_0 as Relative Peak Data size. When using real traces with duplicate items, we calculate the ratio of the insertions to approximate Relative Peak Data Size.

5.2 Experiments on Accuracy

FPR during insertion. EBF *vs.* DBF *vs.* SBF *vs.* Standard BF (Fig. 3): *This experiment shows that the EBF can freely adjust its accuracy (measured by FPR) to keep it accurate.* In this experiment, we first insert 2^{14} items into each kind of Bloom filter, whose initial sizes are the same (2^{18} bits), and detect the FPR of them during the process of inserting items. As shown in the figure, the FPR of standard BF is the worst because it cannot adjust its size. The DBF and SBF (Scalable BF) get worse more slowly but keep increasing along with the RPDS. Our EBF is the best among these Bloom filters because its Bloom filter in the fast memory can expand perfectly. The price is that we consume more slow memory space, but we claim that it is acceptable.

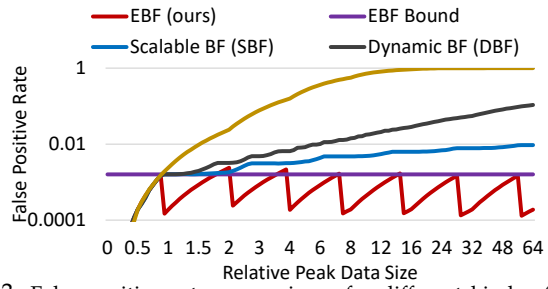


Fig. 3: False positive rate comparisons for different kinds of Bloom filters during the inserting process. The figure is a tendency of FPR as with data size increasing.

Impact of initial memory size on FPR. EBF *vs.* DBF *vs.* SBF (Fig. 4): *This experiment shows that EBF is the best for all initial memory sizes.* When the data size is small, the user wants to keep a small Bloom filter to save memory. When the data size increases significantly, the user wants the data structure to supply the lowest final FPR by the smallest initial memory size. We detect the FPR after expansion. It is shown in the figure that the EBF is the best for all initial memory sizes.

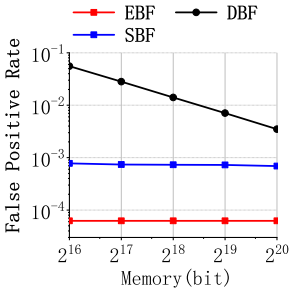


Fig. 4: Impact of initial memory size.

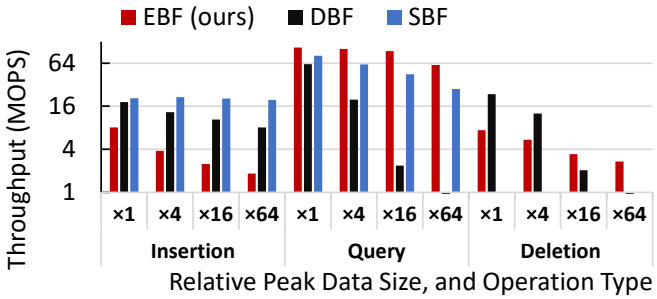


Fig. 5: Processing speed comparison on each operation.

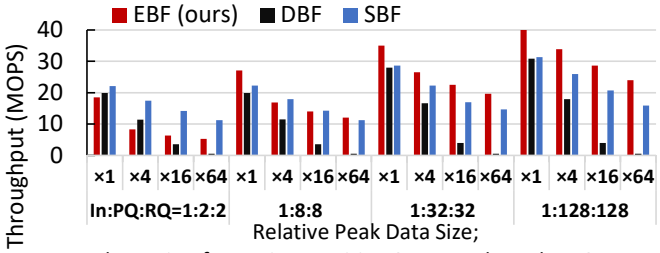


Fig. 6: Overall processing speed comparison.

5.3 Experiments on Processing Speed

Processing speed comparison on each operation. EBF *vs.* DBF *vs.* SBF (Fig. 5): *This experiment shows that the query speed of EBF is much faster than the other two and the deletion speed of EBF is faster when the number of inserted items is large.* In this expriment, we compare the three kinds of Bloom filters' processing speed of different operations. The initial size of each Bloom filter is the same (2^{18}) and the FPR thresholds are guaranteed to be the same. The experiments below of this section have the same settings. We vary the relative peak data size to observe the processing speed of

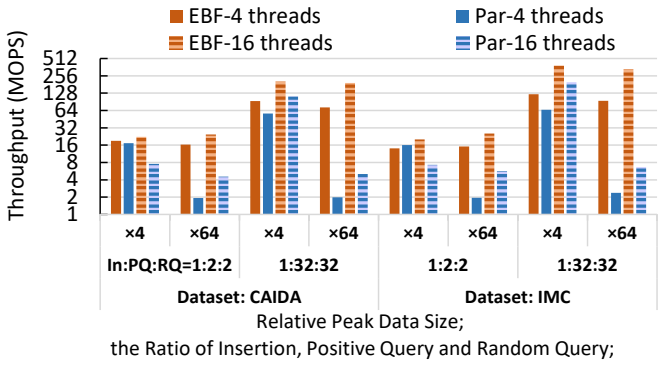


Fig. 7: Multi-thread evaluation.

each Bloom filter. As shown in Fig. 5, the insertion speed of EBF is slower than the other two, but to improve this, we use multi-thread to accelerate the insertion, as is discussed in the next section. Since the query of EBF only needs to query one Bloom filter in the fast memory, its query speed is much faster than the other two. As for the deletion speed, we did not compare that of SBF because it does not support deletion operation. The deletion speed of EBF is faster when the relative peak data size is larger than 16.

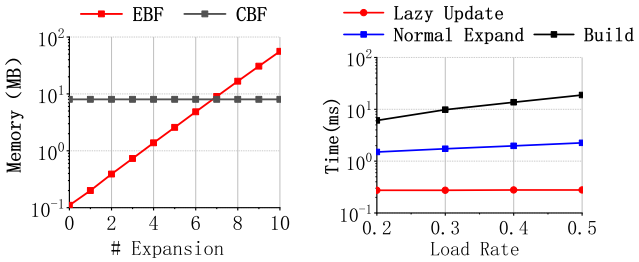
Overall processing speed comparison. EBF *vs.* DBF *vs.* SBF (Fig. 6): *This experiment shows that our EBF is the best query-oriented data structure.* Using real traces (CAIDA), we compare the three kinds of Bloom filters' overall performance, i.e., the speed when the insertions, expansions, and queries are mixed up. We examine the MOPS with the changing ratio of insertion, positive query (i.e., to query the items which has been inserted) and random query (i.e., to query random items, most of which hasn't been inserted). From the figure, we can see that when the ratio is 1:2:2 (After we insert 1 item, we will have 2 positive queries and 2 random queries.) and the relative peak data size is smaller than 64, the MOPS of EBF is slightly lower than the other two. However, in most cases, with the proportion of query increasing, the processing speed of EBF becomes better than the other two.

Overall processing speed comparison using multi-thread technique. EBF *vs.* Par-BF (Fig. 7): *This experiment shows that the multi-thread technique is helpful to accelerate the processing speed.* In this experiment, the multi-thread technique has been used to accelerate the processing speed of EBF. Fig. 7 shows the overall performance with the multi-thread technique. We compare the throughput of EBF and Par-BF using 4 threads and 16 threads. As is shown in the figure, the MOPS of EBF is higher than that of Par-BF.

5.4 Design Choice of Slow Memory Usage

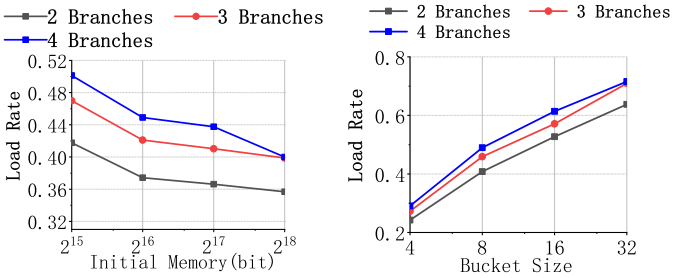
Slow memory consumption. EBF *vs.* CBF (Fig. 8a): A question is why we do not use a large CBF instead of buckets/fingerprints in the slow memory, aiming to help the BF expand. *This experiment (Fig. 8a) shows that EBF uses less memory mostly, but CBF make a better use of memory when its size is nearly the maximum size.* In this experiment, the BF in the fast memory is 2^{12} bits. We allocate 2^{22} counters of slow memory for CBF so that CBF can help the BF expand its size to 2^{22}. We use CAIDA as the dataset. As shown in Fig. 8a, the memory used by EBF varies with the increasing times of expansion, while the memory used by CBF is fixed when it is initialized.

Time cost of expanding operations. Lazy update *vs.* Normal expansion *vs.* Build EBF (Fig. 8b): *This experiment shows the time cost of 3 kinds of operations to expand the EBF.* With the load rate increasing, the number of inserted items increases and the time cost of these operations increases slightly. However, the time cost of lazy update keeps the lowest and that of rebuilding EBF keeps the highest. That's why we optimize the expansion with lazy update.



(a) Slow memory usage. (b) Time cost of memory copy.

Fig. 8: Design choice of slow memory usage.



(a) Load Rate (b) Load Rate

Fig. 9: Load rate comparisons for different numbers of branches in the bucket tree. We vary the Bloom filter size and the bucket size to figure out the relationship between these variables.

5.5 Experiments on the Load Ability in Slow Memory

Load rate *vs.* Bloom filter size (Fig. 9a): *This experiment shows that the load rate gets lower as the size of the Bloom filter gets larger and that more branches can increase the load rate.* In this experiment, 8 hash functions are used to calculate the mapped bits. CAIDA is used and 2^{19} items are inserted into the Bloom filter. As shown in the figure, with the number of branches fixed and the size of the Bloom filter increasing, the load rate gets smaller and smaller. And with the size of the Bloom filter fixed and the number of branches increasing, the load rate gets larger and larger, since we have more buckets to hold the inserted elements.

Load rate *vs.* bucket size (Fig. 9b): *This experiment shows that enlarging the buckets helps increase the load rate.* In this experiment, we compare the load rate with different sizes of the buckets and different numbers of branches in the bucket tree. CAIDA is used and 2^{19} items are inserted into the Bloom filter. As expected, with the number of branches fixed, the load rate increases as the size of the buckets gets larger. And with the size of the buckets fixed, the load rate increases as the number of branches gets larger.

6 CONCLUSION

When Bloom filters are used for dynamic sets, deletion and expansion should be supported. In this paper, we propose the Elastic Bloom filter to support item deletion and expansion at the same time without sacrificing the query efficiency. We propose the Elastic Bloom filter which supports item deletion and expansion at the same time. Our key technique is Elastic Fingerprints. Elastic Fingerprints dynamically absorb and release bits during compression and expansion. Mathematically analysis and experimental results show that the Elastic Bloom filter keeps the advantages of the Bloom filter and is suitable to deal with dynamic sets. The experimental results show that the Elastic Bloom filter can outperform the state-of-the-art in terms of query speed and false positives. We have open-sourced all related source code at GitHub [2].

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (NSFC) (No. U20A20179), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019).

REFERENCES

[1] Hash function source code. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.
[2] Our open souce website [on line]. https://github.com/Dustin-He/ElasticBloomFilter.
[3] The CAIDA UCSD Anonymized Internet Traces Dataset - 20191123. http://www.caida.org/data/passive/passive_dataset.xml.
[4] M. Alasmar, G. Parisis, R. Clegg, and N. Zakhleniu. On the distribution of traffic volumes in the internet and its implications. In IEEE INFOCOM 2019-IEEE conference on computer communications, pages 955–963. IEEE, 2019.
[5] A. Alsirhani, S. Sampalli, and P. Bodorik. Ddos detection system: Using a set of classification algorithms controlled by fuzzy logic system in apache spark. IEEE Transactions on Network and Service Management, 16(3):936–949, 2019.
[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pages 267–280, 2010.
[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.
[8] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. ACM SIGCOMM Computer Communication Review, 36(4):315–326, 2006.
[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In Proc. IEEE HPI, pages 44–51. IEEE, 2003.
[10] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In Proc. ACM SIGCOMM, pages 201–212. ACM, 2003.
[11] C. Estan and G. Varghese. New directions in traffic measurement and accounting, volume 32. ACM, 2002.
[12] B. Fan. Algorithmic Engineering Towards More Efficient Key-Value Systems. PhD thesis, figshare, 2013.
[13] 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 (TON), 8(3):281–293, 2000.
[14] D. Guo, J. Wu, H. Chen, and X. Luo. Theory and network applications of dynamic bloom filters. In Proceedings IEEE INFOCOM 2006, pages 1–12. IEEE, 2006.
[15] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. IEEE Transactions on Knowledge and Data Engineering, 22(1):120–133, 2009.
[16] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng. Locality-sensitive bloom filter for approximate membership query. IEEE Transactions on Computers, 61(6):817–830, 2011.
[17] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta. Energy-efficient neural networks using approximate computation reuse. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE).
[18] H. Lim, K. Lim, N. Lee, and K.-H. Park. On adding bloom filters to longest prefix matching algorithms. IEEE Transactions on Computers, 63(2):411–423, 2012.
[19] Y. Liu, X. Ge, D. H. C. Du, and X. Huang. Par-bf: A parallel partitioned bloom filter for dynamic data sets. International Journal of High Performance Computing Applications, 30(3), 2014.
[20] M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo. Towards collaborative security and p2p intrusion detection. In Proc. IEEE IAW, pages 333–339. IEEE, 2005.
[21] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In 2008 International Symposium on Computer Architecture, pages 453–464, June 2008.
[22] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. IEEE Communications Surveys & Tutorials, 21(2):1912–1949, 2018.
[23] J. H. Mun and H. Lim. New approach for efficient ip address lookup using a bloom filter in trie-based algorithms. IEEE Transactions on Computers, 65(5):1558–1565, 2015.
[24] J. Ni, K. Zhang, Y. Yu, X. Lin, and X. Shen. Privacy-preserving smart parking navigation supporting efficient driving guidance retrieval. IEEE Transactions on Vehicular Technology, 67(7):6504–6517, July 2018.
[25] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 823–829, 2005.
[26] A. Papadopoulos and D. Katsaros. A-tree: Distributed indexing of multidimensional data for cloud computing environments. In Proc. IEEE CloudCom, pages 407–414. IEEE, 2011.
[27] R. Patgiri, S. Nayak, and S. K. Borgohain. Preventing ddos using bloom filter: A survey. arXiv preprint arXiv:1810.06689, 2018.
[28] S. Pontarelli and M. Ottavi. Error detection and correction in content addressable memories by using bloom filters. IEEE Transactions on Computers, 62(6):1111–1126, 2012.
[29] S. Ratnasamy, A. Ermolinskiy, and S. Shenker. Revisiting ip multicast. In Proc. ACM SIGCOMM, pages 15–26, 2006.
[30] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, pages 21–40. Springer-Verlag New York, Inc., 2003.
[31] C. E. Rothenberg, P. Jokela, P. Nikander, M. Sarela, and J. Ylitalo. Self-routing denial-of-service resistant capabilities using in-packet bloom filters. In Proc. EC2ND, pages 46–51. IEEE, 2009.
[32] M. Sarela, C. E. Rothenberg, T. Aura, A. Zahemszky, P. Nikander, and J. Ott. Forwarding anomalies in bloom filter-based multicast. In Proc. IEEE INFOCOM, pages 2399–2407. IEEE, 2011.
[33] U. Sattar, T. Naqash, M. R. Zafar, K. Razzaq, and F. B. Ubaid. Secure dns from amplification attack by using modified bloom filters. In ICDIM 2013, 2014.
[34] J. Wei, H. Jiang, K. Zhou, and D. Feng. Dba: A dynamic bloom filter array for scalable membership representation of variable large data sets. In 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems.
[35] T. Wolf. A credential-based data path architecture for assurable global networking. In Proc. IEEE MILCOM 2007, 2007.
[36] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie. A scalable bloom filter for membership queries. In IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference, pages 543–547. IEEE, 2007.
[37] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li. A shifting bloom filter framework for set queries. Proceedings of the VLDB Endowment, 9(5):408–419, 2016.
[38] H. Yu and R. Mahapatra. A memory-efficient hashing by multi-predicate bloom filters for packet classification. In Proc. IEEE INFOCOM, 2008.
[39] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: Bloom filter forwarding architecture for large organizations. In Proceedings of the 5th international conference on Emerging networking experiments and technologies, pages 313–324, 2009.