AUT Journal of Electrical Engineering

AUT J. Elec. Eng., 51(2) (2019) 187-200
DOI: 10.22060/eej.2019.16370.5287

A Cuckoo Filter Modification Inspired by Bloom Filter

Ha. Sasaniyan Asl[1,*], B. Mozaffari Tazehkand[2], M.J. Museviniya[2]

[1] MSc,Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran
[2] PhD,Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran

ABSTRACT: Probabilistic data structures are so popular in membership queries, network applications, and databases and so on. Bloom Filter and Cuckoo Filter are two popular space efficient models that incorporate in set membership checking part of many important protocols. They are compact representations of data that use hash functions to randomize a set of items. Being able to store more elements while keeping a reasonable false positive probability is a key factor of design. A new algorithm is proposed to improve some of the performance properties of Cuckoo Filter such as false positive rate and insertion performance and solve some drawbacks of the Cuckoo algorithm such as endless loop. Main characteristic of the Bloom Filter is used to improve Cuckoo Filter, so we have a smart Cuckoo Filter which is modified by Bloom Filter so called as SCFMBF. SCFMBF uses the same table of buckets as Cuckoo Filter but instead of storing constant Fingerprints, it stores Bloom Filters. Bloom Filters can be accumulated in the table's buckets which leads to higher insertion feasibility. We also address the endless loop problem of Cuckoo Filter that means an inserted item is stuck in an iterative process of finding an empty bucket, so a smart algorithm is designed which not only solves endless loop problems but also prevents insertion failure. Hence our smart algorithm prevents double checking of a bucket and avoids making loops. Consequently, the capacity of SCFMBF is improved significantly. Results of comparison with Cuckoo Filter shows that false positive probability of SCFMBF method is enhanced compared to Cuckoo Filter.

Review History:

Received: 2019-05-20
Revised: 2019-09-17
Accepted: 2019-09-17
Available Online: 2019-12-01

Keywords:

Bloom Filter
Cuckoo Filter
endless loop
Hash Function
set membership

1. INTRODUCTION

Bloom Filter is a space efficient probabilistic data structure which is used to represent a set and perform membership queries [1] to query whether an element is a member of a set or not. Bloom Filter was first proposed by B. H. Bloom [2] as a membership query structure, and provided the framework for many other researchers. A Bloom Filter occupies a negligible space for representation of members compared to entire sets. Representation of sets with a limited data structure as the Bloom Filter, leads to false positives. False negatives are not possible in Bloom Filter, it means when Bloom Filter shows an item is not a member of the set, it is definitely not in the set. On the other hand, with a certain probability, Bloom Filter wrongly declares an item is a member of the set, this probability is an important factor in the filter design which is called False Positive Probability (FPP). The structure and properties of Bloom Filter is described in section 3.1. Bloom Filters are used in applications that need to search a member in large sets of items in a short time such as spell checking, network traffic routing and monitoring, data base search, differential file updating, distributed network caches, textual analysis, network security and iris biometrics. Some of the popular applications are described below:

- Spell checking: Determination of whether a word is a valid word in its language, is done by creating Bloom Filter for all possible words of a language and checking a word against the Bloom Filter [3].
- Routing: if the network is in the form of a rooted tree with nodes holding resources and a node receives a request for resource, it checks its unified list to ascertain if it has a way of routing that request to the resource [4]. False positives in this case may forward the routing request to an incorrect path. In order to solve this, each node in the network keeps an array of Bloom Filter for each adjacent edge.
- Network Traffic: Bloom filters are widely used to reduce network traffic and are used in caching proxy servers on the World Wide Web (WWW). Bloom Filters are used in web caches to efficiently determine the existence of an object in cache. Bloom Filters are also used as cache digest. A cache digest contains information of all cache keys with lookup capability. By checking a neighbor cache, a cache can determine with certainty if a neighboring cache does not hold a given object [5].

*Corresponding author's email: sasaniyan@tabrizu.ac.ir

A Bloom Filter is very much like a hash table in that it will use a hash function to map a key to a bucket. However, Bloom Filter does not store that key in the array (bucket), it simply sets some bits to 1, to mark the bucket as filled. So, many keys might map to same filled bucket, creating false positives. A Bloom Filter also includes a set of k hash functions with which we hash incoming values. These hash functions must all have a range of 0 to m - 1. If these hash functions match an incoming value with an index in the bit array, the Bloom Filter will make sure the bit at that position in the array is 1. In order to adapt Bloom Filter to the variety of applications, there is need to modify Bloom Filter to be more effective. Several papers have tried to reduce false positive probability [6], provide member deletion for Bloom Filter [7], handle large data sets [8], make scalable Bloom Filter for dynamic sets [9], find the similarity of two sets [10] and reduce the size of Bloom Filter [11]. We are going to introduce some famous modifications of Bloom Filter as examples in the Related Work section.

The Cuckoo Filter is a minimized hash table that uses Cuckoo hashing to resolve collisions. It minimizes its space complexity by only keeping a fingerprint of the value to be stored in the set. Much like the Bloom Filter that uses single bits to store data, the Cuckoo Filter uses a small f-bit fingerprint to represent the data. The value of f is decided on the ideal false positive probability the programmer wants. The Cuckoo Filter has an array of buckets. The number of fingerprints that a bucket can hold is referred to as b. They also have a load which describes the percent of the filter they are currently using. So, a Cuckoo Filter with a load of 75% has 75% of its buckets filled. This load is important when analyzing the Cuckoo Filter and deciding if and when it needs to be resized.

In this article, we proposed a smart Cuckoo Filter to solve some of shortcomings of the previous methods described in the Related Work section. Being smart is about detecting and getting out of endless loops. Since Cuckoo Filter is a fixed data structure, if the load factor is exceeded from its predetermined value related to the tolerable error probability, the insertion fails. Load factor is the percentage that shows the Cuckoo table's fullness. To avoid data loss caused by removals or new insertions, a modified Cuckoo Filter is created by our Cuckoo Support Algorithm (CSA). Our modification of Cuckoo Filter is inspired by the standard Bloom Filter. Standard Bloom Filter makes the final Bloom Filter array by calculating the union of Bloom Filters which is done by logical OR operation. Therefore our modified Cuckoo Filter has more capacity by allowing new insertions even when the Cuckoo table is full.

Cuckoo Filter and Bloom Filter are both popular probabilistic data structures. Although they are built differently, each of them have special useful characteristics and we want to benefit from both to reach a more powerful membership query structure. Unlike the Standard Bloom Filter, Cuckoo Filter has removal capability and we build the protocol on Cuckoo Filter first. When Cuckoo table is close to fullness, there is hardly enough space for new insertions. We solve this space deficiency of Cuckoo Filter by making a change in Cuckoo Filter's structure. Bloom Filter is used as the former fingerprint of data and a new phase is added to Cuckoo Filter algorithm, that leads to higher Cuckoo table capacity and also impressively lower false positive probability. Cuckoo Filter has the problem of endless loops, that means when it cannot find an empty bucket for the new insertion, it checks the same buckets again and again. We designed a new algorithm to make Cuckoo Filter smart not only to detect endless loops but also get out of them. Standard Bloom Filter, (as mentioned in Related Work section) has many extensions that makes it suitable for different applications. Standard Bloom Filter satisfied our design goals that is why we didn't use any of the Bloom Filter extensions. Comparison results demonstrates our protocol's being successful in lowering the key design factor of these filters, false positive probability, to a low negligible amount. By changing the filter elements further, FPP can even fall to lower amounts.

We will exclusively compare our proposed filter with the Cuckoo Filter. With the changes we have made to Cuckoo Filter, a new filter is made that we name it Smart Cuckoo Filter Modified by Bloom Filter (SCFMBF). We briefly explain standard Bloom Filter and Cuckoo Filter characteristics then relate it to our protocol SCFMBF.

In the following, the article outline is summarized. Section 2 is dedicated to the related work that mainly introduces papers about different extensions of Bloom Filter and Cuckoo Filter. We are going to explain Bloom Filter and Cuckoo Filter concepts further in the section 3, because they are the base of our proposed method. In section 4, our protocol is introduced and the designed algorithms are explained. In sections 5 and 6 the calculation results and the Fig.s of comparison are considered, we have compared our protocol with the Cuckoo Filter to check its efficiency. Section 7 is a summarized conclusion about our work.

2. RELATED WORK

Laufer et al. [6] introduces Generalized Bloom Filter, that employs two groups of hash functions $\{g_1, \ldots, g_{k0}\}$, $\{h_1, \ldots, h_{k1}\}$. When inserting an element into the bit vector, the g hash functions set the bits for the item and the h hash functions reset the bits. False positive happens when the $g_1(x), \ldots, g_{k0}(x)$ are all 0 and the bits $h_1(x), \ldots, h_{k01}(x)$ are all 1. Fan et al. [7] suggested Counting Bloom Filter (CBF) to add deletion possibility to the standard Bloom Filter. CBF puts counters instead of single bits in the Bloom Filter array and increments the counters when new set member is inserted. Split Bloom Filter by Xiao et al. [8] employs a constant $s \times m$ bit matrix for set representation, where s is a pre-defined constant based on the estimation of maximum set cardinality and m denotes Bloom Filter array length. Guo et al. [9] suggested Dynamic Bloom Filter as an alternation to Bloom Filter that dynamically creates new filters when needed. When the false positive rate of a Bloom Filter is rising fast, it switches to a new Bloom Filter instead. Geravand et al. [10] proposed Matrix Bloom Filter that contains N rows of Bloom Filter with m bits used to find out the similarities between two documents. Matrix BF just executes the bitwise AND operations between the two rows. Thereby, the similarity is measured by counting the number of 1s in the resultant bit array. Compressed Bloom Filter by Mitzenmacher [11] is for transmission size optimization of the Bloom Filter. The main idea of this protocol is based on changing the way bits are distributed in the filter. Counting Bloom Filter chooses the number of hash functions in a way that the Bloom Filter's entries have a smaller probability than ½. d-left counting Bloom Filter proposed by Bonomi et al. [12] is equivalent to a Counting Bloom Filter by a factor of two or more space. d-left counting Bloom Filter divides a hash table into d sub-tables so d hash functions are needed. Hierarchical Bloom Filter proposed by Shanmugasundaram et al. [13] is a multi-level filter. When a string is inserted it is first broken into blocks which are inserted into the filter hierarchy starting from the lowest level. In that way, sub-string matching is supported. Cohen et al. suggested [14] Spectral Bloom Filter to support frequency queries by counters to store the frequency values. Matsumoto et al. proposed [15] Adaptive Bloom Filter, a modification of counting Bloom Filters for the cases that large counters are needed. Almeida proposed Scalable Bloom Filter [16] that adds slices of Bloom Filter as the set grows. Weighted Bloom Filter is designed by Bruck et al. [17]. It is a Bloom Filter that changes the number of hash functions according to element query popularity, so it incorporates the information on the query frequencies and the membership likelihood of the elements. Quotient Filter is suggested by Bender et al. [18] that is a compact hash table in which the table entries contain only a portion of the key plus some additional meta data bits. In a quotient filter a hash function generates a fingerprint. Some of the least significant bits is called the remainder and the most significant bits are called the quotient. The remainder is stored in the quotient's corresponding slot. We used the Standard Bloom Filter in our proposed method, because by combining Cuckoo and Bloom Filter we got a satisfying result.

Cuckoo Filter was first introduced by Fan et al. [19], Cuckoo Filter is a minimized hash table that stores a summary of a set of inputs. Cuckoo Filter meets the purpose of set membership and we take a close look at it. Fan et al. [19] proposed Cuckoo Filter that is a compact variant of a Cuckoo hash table that stores only fingerprints that are driven by hash function from the input items for each item insertion. A hash table is a particular implementation of a dictionary whose entries are called buckets. For inserting an item, a hash function is used to select which bucket to store the item. The structure and properties of Cuckoo Filter is described in section 3.2. Applications of Cuckoo Filter are a lot similar to applications of Bloom Filter but Cuckoo Filters are more suitable for applications that need to store many items and keep low false positive rate. One of the applications of Cuckoo Filter is represented by Grashofer et al. [20] that used Cuckoo Filter for network security monitoring for processing high band-width data streams. A common pattern is to use probabilistic data structures upstream, to filter out a vast number of irrelevant queries. There are also different modifications of Cuckoo Filter to make it more effective.

Mitzenmacher et al. [21] designed Adaptive Cuckoo Filter. It does not use partial-key Cuckoo hashing, the buckets an element can be placed in are determined by hash values of the element, and not solely on the fingerprint. The filter uses the same hash functions as the main Cuckoo hash table. The element and the fingerprint are always placed in corresponding locations. Sun et al. [22] proposed Smart Cuckoo Filter. The idea behind Smart Cuckoo is to represent the hashing relationship as a directed pseudoforest and use it to track item placements for accurately predetermining the occurrence of endless loop. The aforementioned method doesn't not completely solve the endless loop problem it just detects it and prevents getting stuck in it.

Some of the existing schemes for endless loop problems are Cuckoo Hashing with a stash (CHS). Kirsch et al. [23] proposed CHS for solving the problem of endless loops by using an auxiliary data structure as a stash. The items that introduce hash collisions are moved into the stash. For a lookup request, CHS has to check both the original hash table and the stash, which increases the lookup latency. Erlingsson et al. [24] proposed Bucketized Cuckoo Hash Table (BCHT) that allocates two to eight slots into a bucket, in which each slot can store an item, to mitigate the chance of endless loops, which however results in poor lookup performance due to multiple probes. Fan et al. [25] proposed MemC3 which uses a large kick-out threshold as its default kick-out upper bound, which possibly leads to excessive memory accesses and reduced performance.

In many applications, process of data storage with high accuracy can have significant impact on some strategy decisions. Some of existing storage solutions use BF that uses huge space. Singh et al. [26] proposed Fuzzy Folded BF that is an effective space-efficient strategy for massive data storage, fuzzy operations are used to accommodate the hashed data of one BF into another to reduce storage requirements. It uses only two hash functions to generate k hash functions.

In peer to peer (P2P) distributed storage systems, maintaining the rule of structured overlay without imposing any additional overhead to reconfigure index between saved information and its stored node is important. It cannot be achieved by the traditional P2P techniques. Sasaki et al. [27] uses Distributed Bloom filter Table (DBFT), with physical change of node, node's ID remains unchanged. DBFT is the two-layered structured which indexes by Bloom Filter-based parameters. It reduces overhead, which is imposed by maintaining the function of P2P networks. Rothenberg et al. [28] introduced the Deletable Bloom Filter (DlBF) that is a kind of BF. The DlBF keeps record of the bit regions where collisions happen. This allows safe element removal. Reviriego et al. [29] talk about the application of Bloom Filter. It is shown that BFs can be used to detect and correct errors in their associated data set. This allows a synergetic reuse of existing BFs to also detect and correct errors, it is efficient solution to mitigate soft errors in applications which use CBFs. Lim et al. [30] designed Ternary Bloom Filter (TBF) as an alternative to Counting BF for performance improvement. TBF allocates the minimum number of bits to each counter and includes more number of counters instead to reduce false positive probability. TBF provides much lower false

Table 1. Parameter Definitions

$S=\{x_1, x_2, \ldots, x_n\}$	**Input set**
n	number of items
mb	Bloom Filter array length
k	number of hash functions
h	hash function
FPP	False Positive Probability
f	fingerprint **length in bits**
α	load factor (0≤α≤1)
b	number of entries per bucket
m	number of buckets
c	average bits per item
f = fingerprint(x)	
i_1 = hash(x)	
$i_2 = i_1 \oplus$hash(f)	
fingerprint = h(x_i)	
counter	**C** = $j_1 \| j_2$
z	number of 1s in the finger print
r	maximum number of OR operation

positive rates than the CBF. Ficara et al. [31] use Huffman code to improve standard CBFs in terms of fast access and limited memory consumption (up to 50% of memory saving). It allows an easy lookup since most processors provide an instruction that counts the number of bits set to 1 in a word.

There are also some other papers about application of Bloom Filter in network such as Space-Code Bloom Filter by Kumar et al.[32], and Fast Dynamic Multiple-Set Membership Testing by Hao et al. [33]. Duan et al. [29] proposed a distributed public cloud storage system that allows users to store files named as CSTORE. It is based on a three-level mapping hash method. In order to avoid duplicated data storage, CSTORE adopts Bloom Filter algorithm to check whether a file block is in the meta data set. Geravand et al. [35] designed an MBF-based document detection system. They used Matrix Bloom Filter to prevent plagiarism on the internet. Matrix Bloom Filter consist of some rows of Standard Bloom Filters to support more insertions.

One of the simple and important techniques to control the integrity of data in a data set is check summing data in parallel or serial forms. A parallel data set check summing approach named as *fsum* is proposed by Xiong et al. [36]. They at first broke the files into chunks with reasonable sizes and then chunk-level based checksums are calculated in parallel form. In final step a single data set level checksum is obtained using a Bloom Filter. To improve the performance of Bloom Filters, fast Bloom Filters have been proposed by Qiao et al. [37] as named *Bloom-1* which have a reduced query overhead with an acceptable higher false positive rate for a known memory size. Reviriego et al. [38] evaluated a correct analysis of *Bloom-1* and corresponding exact formula about false positive probability is calculated by them.

Multidimensional Bloom Filter named *Bloofi* is introduced by Crainiceanu et al. [39] to reduce the search complexity of membership queries when the number of Bloom Filters increased. Big data management with widespread applications in IoT environment is unavoidable, therefore today, efficient storage media, high speed processing algorithms, and accessing of bulky data sets in short times are necessary. Recently, a variant of scalable Bloom Filter named as Accommodative Bloom Filter (ABF) is introduced by Singh et al. (2018a) to solve these requirements. The classic and standard Bloom Filter analysis using -transform confirmed the known results and new issues also are obtained by Singh et al. [40] Recently, an overview of Bloom Filter and its variants with optimization methods are deliberated by Grandi [41] with performance and generalization review in more details.

3. DEFINITION

At first, it is necessary to declare all the notations which are used in this article about the Bloom, Cuckoo Filters and our proposed method. Table 1 denotes these definitions.

3-1- Bloom Filter

Bloom Filter is a compact approximate data structure which enables membership queries. For the set $S=\{x_1, x_2, \ldots, x_n\}$ with n elements, a Bloom Filter of size mb is constructed. All the mb bits in the vector are initialized to 0. A group of k

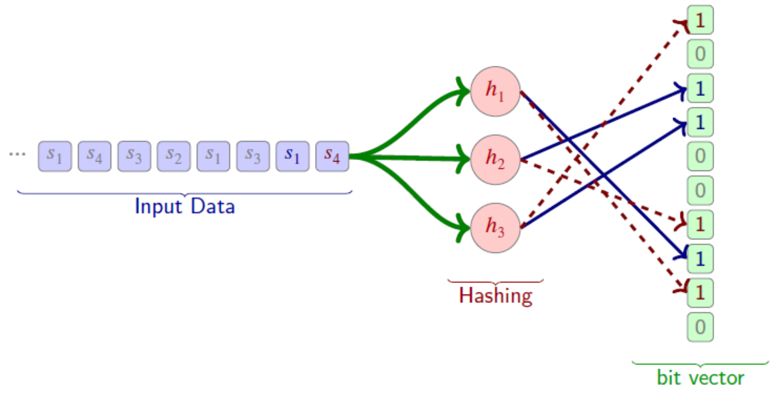

Fig. 1. Bloom Filter example

independent hash functions are employed to randomly map each set member into k positions. If any bit at the k hashed positions of the element equals 0, it means this element does not belong to the set. Otherwise, the Bloom Filter infers that the element is a member of the set with a probability of false positive. Bloom Filter has two main operations: Insertion and Look up. Insertion simply adds an element to the set. Removal is impossible without introducing false negative, but extensions to the Bloom Filter are possible that allow removal e.g. Counting Bloom Filters.

To add an element to the Bloom Filter, we simply hash it a few times and set bits in the bit vector at the index of those hashes to 1. To query for an element, feed it to each k hash functions to get k array positions, if any of the bits at these positions is 0, the element is definitely not in the set, if it were then all the bit would have been set to 1 when it was inserted. If all are 1 then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements resulting in a false positive. In Bloom Filter there is no way to distinguish between the two cases. An example of Bloom Filter construction is given in Fig. 1 for mb=10, k=3.

3-2- Cuckoo Filter

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each positions of the hash table, often called a bucket, can hold an item and is named by an integer value starting at 0. The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of bucket numbers between 0 and m-1. Once the hash values have been computed, we can insert each item into the hash table at the calculated positions. When we want to search for an item simply use the hash function to compute the bucket number for the item and then check the hash table to see if it is present. Standard Cuckoo hash tables have been used to provide set membership information. Cuckoo Filter has two hash function $h_1(x)$ and $h_2(x)$ that points to two different positions in the hash table.

In Cuckoo hashing, each item is hashed by two different hash functions, so that the value can be assigned to one of two buckets. The first bucket is tried first. If there's nothing there, then the value is placed in bucket 1. If there is something there, bucket 2 is tried. If bucket 2 if empty, then the value is placed there. If bucket 2 is occupied, then the occupant of bucket 2 is evicted and the value is placed there. In the process of Cuckoo Filter, we may encounter getting stuck in endless loops. Endless or infinite loop may happen because of the Cuckoo table's being occupied more than a calculated threshold. Since the algorithm cannot find an empty bucket for the insertion, it keeps checking other buckets iteratively and may never be able to find an empty bucket so the insertion fails.

For item insertion there are two positions that the algorithm checks for emptiness. When two items hash to the same bucket, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash functions are perfect, collisions would never occur, however since this is not possible, collision resolution becomes a very important part of hashing. Cuckoo's method for resolving collisions is looking into the hash table and trying to find another open bucket to hold an item that causes the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the buckets until we encounter the first bucket that is empty. If in this search we return to the first bucket we have begun with, we are trap in an endless loop. To make a space efficient Cuckoo Filter and reduce the hash table size, each item is first hashed into a constant sized fingerprint before it is inserted into hash table.

As shown in Fig. 2, we use an example to illustrate the insertion process in the conventional Cuckoo hashing. In the Cuckoo graph, the start point of an edge represents the actual storage position of an item and the end point is the backup position. For example, the bucket $T_2[1]$ storing Item *b* is the backup position of Item *a*. We intend to insert the item *x*, which has two candidate positions $T_1[0]$ and $T_2[5]$. There exist three cases about inserting Item *x* [22]:

• Two items (a and b) are initially located in the hash tables as shown in Fig. 2(a). When inserting Item x, one of x's two candidate positions (i.e., $T_2[5]$) is empty. Item x is then placed in $T_2[5]$ and an edge is added pointing to the backup position ($T_1[0]$).

• Items c and d are inserted into hash tables before Item x, as shown in Fig. 2(b). Two candidate positions of Item x are

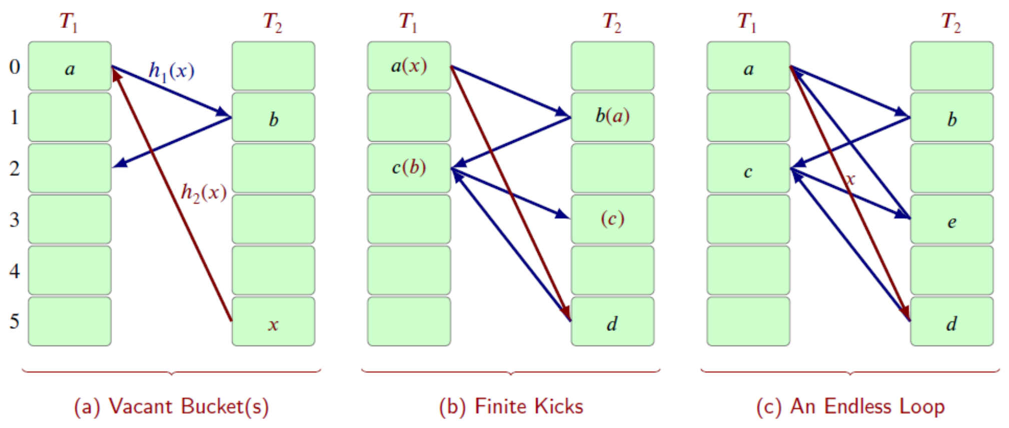

Fig. 2. The conventional Cuckoo hashing data structure [22]

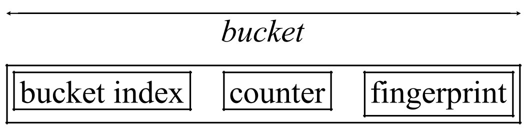

Fig. 3. bucket structure

occupied by Items a and d respectively. We have to kick out one of occupied items (e.g., a) to accommodate Item x. The kicked-out item (a) is then inserted into its backup position (T_2[1]). This procedure is performed iteratively until a vacant bucket (T_2[3]) is found in the hash tables. The kick-out path is x→a→b→c.

• Item e is inserted into the hash tables before Item x, as shown in Fig. 2(c). There is no vacant bucket available to store Item x even after substantial kick out operations, which results in an endless loop. The Cuckoo hashing has to carry out a rehashing operation.

4. PROPOSED METHOD

SCFMBF is smart because it allows deletion and insertion of items while keeping the false positive probability at an acceptable rate. It is based on Cuckoo Filter because of its good performance and popularity. It uses of a new algorithm to make Cuckoo Filter smart to detect endless loops and get out of them. That will lead to a higher Cuckoo table capacity. SCFMBF uses a Cuckoo Support Algorithm (CSA) for solving kicking problem in Cuckoo Filter therefore it improves insertion performance. SCFMBF is divided into three algorithms: a modified Cuckoo Filter algorithm, endless loop algorithm and Cuckoo Support Algorithm (CSA).

We have briefly explained standard Bloom Filter and Cuckoo Filter characteristics and formulas and then relate it to our method SCFMBF. In contrary to the Bloom Filter that uses a bit array that is an array of single-bit buckets. In SCFMBF we use extended buckets like Cuckoo Filters. Each bucket holds a fingerprint of the element, and a counter (Cuckoo counter j_2 and Cuckoo sign j_1) and the bucket's index. Fig. 3, shows SCFMBF array structure.

We take the idea of Cuckoo Filter in which each item is hashed into a p-bit fingerprint that is divided into two parts: a bucket index and a value part (finger print) to be stored. In Cuckoo Filter if bucket i (called the primary) is full then the Cuckoo Filter attempts to store f in bucket i ⊕ h(f), (⊕ is the logical XOR operation), where h is a hash function. If both buckets are full, then the Cuckoo Filter kicks that item out of one of the two buckets, moving it to its alternate location. One of the drawbacks of Cuckoo Filter is falling into endless loops while looking for empty bucket for item insertion. There are some designed algorithms for preventing endless loops from happening. That's the reason we need a modification for Cuckoo Filter. Solving the problem of endless loop is crucial because CSA is implemented right after facing that problem.

In the counter there are two partitions: Cuckoo sign (j_1) and Cuckoo counter (j_2). Cuckoo counter is related to the Cuckoo Support Algorithm (CSA) which is going to be described. Since Cuckoo Filter's size is fixed and predetermined, a load factor (α) that shows the fullness of the Cuckoo table is specified according to our acceptable false positive probability. Whenever load factor reaches its maximum value, insertion fails. In order to prevent data loss, we designed the Cuckoo support algorithm. In this case if we fall into endless loop, we need to realize it and stop it in a way.

4-1- Modified Cuckoo Filter

According to the endless loop problem of Cuckoo Filter, we decided to change Cuckoo Filter to solve this, using a new problem-solving algorithm, we name it the Modified Cuckoo Filter (MCF). Structure of the Modified Cuckoo Filter is demonstrated in Fig. 4.

In our protocol, the fingerprint of the input item is inserted to the table (like Cuckoo Filter) but this fingerprint is the calculated Bloom Filter of the input item.

Cuckoo sign: We use Cuckoo sign in Cuckoo Filter as a simple bit for endless loop problem solving by bucket

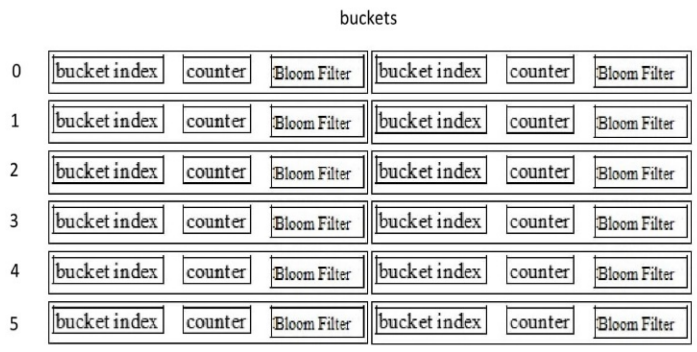

Fig. 4. structure of modified Cuckoo Filter

Algorithm 1. sign bit (SB) algorithm

```
While item x is inserted suppose $i_1$'s and $i_2$'s bucket is full do
  Check $i_1$'s corresponding bucket;
  if $i_1$'s Cuckoo sign is 0 then
  set it to 1;
  end
  Check $i_2$'s corresponding bucket;
  if $i_2$'s Cuckoo sign is 0 then
  set it to 1;
  end
  Put x into $i_2$'s bucket and kick the saved fingerprint to its $i_{v1}$ bucket;
  Check $i_{v1}$'s cuckoo sign;
  if $i_{v1}$'s Cuckoo sign is 0 then
    Set it to 1;
  else
    Check the load factor α;
    if α is less than the desired value then
        delete the input;
    else
        Perform CSA algorithm;
    end
  end
  reset sign bits of the buckets;
end
```

checking. The Cuckoo sign is used to show whether a bucket is checked for emptiness. While our algorithm is looking for an empty bucket in the insertion, by passing and checking each bucket, its Cuckoo sign is set to 1. And after each item insertion, Cuckoo signs are reset. An endless loop is when we keep checking the same buckets over and over again, when a bucket is checked for the second time, the Cuckoo sign is already set, so the algorithm finds out it has been trapped in a loop. In this situation the current load factor is checked, if the load factor is less than our desirable value, the current insertion is canceled (because no empty bucket is found while the table is not full), but if it has reached that value, then the CSA algorithm is performed.

Algorithm 1 shows the role of sign bit in the proposed protocol. When item x is inserted, i_1 and i_2 (the indexes of the buckets) are calculated by hash functions. The corresponding buckets are checked for emptiness and their Cuckoo sign bit is set to 1. x is put into i_2's bucket and the saved fingerprint is kicked to its other possible bucket i_{v1}. If the sign bit of i_{v1} is 0, it is set to 1, and if the bucket is full, the old fingerprint is kicked to its other possible bucket i_{v}. This process is repeated till whether an empty bucket is found (end of the algorithm) or we reach a bucket that has been checked before which means endless loop (its Cuckoo sign is 1). In this level there are two cases: we check the load factor of the table, if it is less than the desired value, the table is not full and the input is deleted (insertion failure just like the original Cuckoo Filter algorithm), but if the load factor has reached the desired value, the table is full and the Cuckoo support algorithm must be performed.

Cuckoo support algorithm (CSA): When the Cuckoo table is filled up to the determined load factor, Cuckoo Filter algorithm is not followed. When a new item faces a full bucket, sum of the new item's fingerprint and the bucket value is calculated (sum is done by OR operation) and is placed in the bucket and Cuckoo counter is incremented by 1. When the Cuckoo counter is non zero, it means CSA is performed and in SCFMBF look up algorithm we will return to the usefulness of this counter. Again, Algorithm 2 indicates the details of CSA method.

Algorithm 2. Cuckoo support algorithm (CSA)

```
1 Want to insert fingerprint(x) to the table;
2 Goto $i_1$'s bucket and take the saved fingerprint;
3 Sum up fingerprint(x) and the already saved fingerprint by OR operation;
4 Put the OR result in the $i_1$'s bucket;
5 Increment the Cuckoo counter by 1;
```

Algorithm 3. Insertion algorithm

```
Insert data block $x_i$;
Hash the block and calculate its fingerprint $f_i$;
Derive two indices from the hash and fingerprint: $i_1$ and $i_2$;
if the table is not full then
    Check the derived indices;
    *: if empty then
        Insert the block's fingerprint into the corresponding bucket
        Reset $j_1$;
    else
        set $j_1=1$;
        Check current item's alternative indices;
        if $j_1 = 0$ in the alternative buckets then
            Goto *
        else
            a bucket with $j_1=1$ is reached, endless loop detected;
        delete $f_i$;
        end
    end
else
        CSA: substitute $B(i_1) + f(x_i)$ in the $B_i$ Bucket;
        Increment $j_2$;
End
```

Algorithm 2 is the second part of insertion process. The Cuckoo table is already full and the i_1 index is calculated for newly inserted item. The protocol refers to the bucket i_1 and takes the already saved fingerprint, performs an OR operation between the two fingerprints and saves the result in the corresponding bucket then increments the Cuckoo counter.

SCFMBF Insertion: Main part of insertion is like the Cuckoo Filter unless we reach an endless loop that needs the endless loop algorithm to be implemented. That may lead us to CSA algorithm. Algorithm 3 shows the details of insertion method in SCFMBF scenario.

To insert a new item into the Cuckoo table, the fingerprint of it is calculated (the Bloom Filter). Then the two indexes of the buckets are derived by the hash functions. If the Cuckoo table is not full, the corresponding buckets are checked for emptiness, whenever an empty bucket is found, the fingerprint f is put in there, but when the buckets are full, the saved fingerprints are kicked to their possible buckets, their cuckoo sign is set to 1, this process is repeated till an empty bucket is found or an insertion failure happens (endless loop j_1=1 is detected). If the Cuckoo table is full, the first possible bucket for the input is checked, the old finger print $B(i_1)$ is retrieved and an OR operation is performed on the two fingerprints. The result of the operation is saved in the current bucket. Cuckoo counter j_2 is incremented as a symbol of CSA algorithm's being performed.

Algorithm 4. Look up algorithm

```
f = fingerprint(x);
 $i_1$ = hash(x);
 $i_2 = i_1 \oplus$hash(f);
check $i_1$'s Cuckoo counter;
if $j_2$ =0 then
        Goto Cuckoo algorithm;
else
        Goto CSA algorithm;
end
Cuckoo algorithm: if bucket($i_1$) or bucket($i_2$) has f then
    return True;
else
     return False;
end
     CSA algorithm: Get the saved array;
     Do OR operation on the fingerprint and the saved array;
     if the result is the same as the saved array then
         return True;
     else
         return False;
     end
```

Algorithm 5. Delete algorithm

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 $\oplus$hash(f);
if Lookup(f) then
  if $j_1 > 0$ then
    Get the saved array;
      Decrement $j_2$ by one;
      return True;
    end
 if $j_1$=0 then
       Remove a copy of f from this bucket;
       return True;
  end
else
 return False;
end
```

SCFMBF Look up: As shown in Algorithm 4, main part of the lookup procedure is like the Cuckoo Filter. We want to check whether x belongs to the set S or not. The fingerprint of x is calculated. Two indexes i_1 and i_2 are calculated. Then i_1's bucket is visited. If the Cuckoo counter is zero, it means the CSA algorithm has not been implemented for that set, and we look for the original form of fingerprint(x). If they match, with a false positive probability (Cuckoo FPP), x belongs to the set. If the Cuckoo counter j_2 is not zero, we realize the CSA has been performed. We take the idea of the Bloom Filter: if the positions of 1s in the fingerprint(x) matches the saved array, it means with a probability, x belongs to the set. Because with the OR operation, positions of 1s don't change and remain the same. The false positive probability of the lookup procedure (set membership verification) is the total FPP of this algorithm that is going to be described.

To check whether x belongs to the set (is in the Cuckoo table), first the fingerprint of x is calculated, and then the indices i_1 and i_2 are derived. i_1's Cuckoo counter is checked to see if CSA has been performed or not. If j_2 =0, original Cuckoo algorithm has been performed then bucket i_1 and bucket i_2 are checked. If the exact fingerprint of x is found, algorithm returns True, which means x belongs to the set, otherwise x is not in the set. If $j_2 \geq 1$, saved array in bucket i_1 is retrieved, an OR operation is performed on f_i and the saved array, if the number of 1s and the positions remain the same in the saved array, it means it contains that fingerprint, then the algorithm returns True, but if the OR operation adds new 1 positions to the saved array, x doesn't belong to the set.

SCFMBF Deletion: Delete algorithm is for removing an item from the table and is examined in Algorithm 5.

Sometimes there is need to remove an item from the set. The fingerprint and the indices of that item is calculated. The protocol calls the look up algorithm to make sure the item

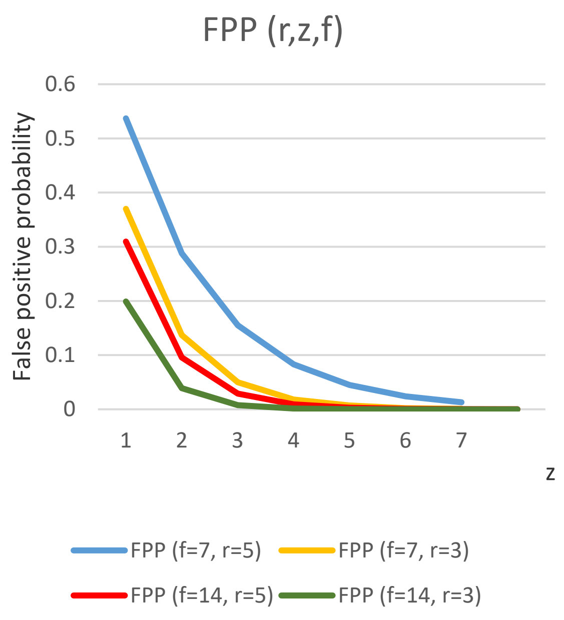

Fig. 5. False Positive Probability of Cuckoo Support Algorithm

belongs to the set, if not it returns False. If j_2=0, it means no OR operation has been performed and a copy of f is removed from the bucket. If $j_2 \geq 1$, it means there are more than 1 fingerprints aggregated in that bucket. Since OR operation is not reversible, we can't remove or change the saved content but we decrement j_2 by one to show deletion.

5. PROBABILITY OF INSERTION FAILURE

False positive error probability of the modified Cuckoo Filter is the same as the original Cuckoo Filter because the main structure is unchanged and we have only added a counter part to the main structure.

5-1- False error probability of CSA algorithm

Let f denote the number of bits in the fingerprint. When inserting an element into a full bucket, the probability that a certain bit is not set to one is:

$$1-\frac{1}{f} \tag{1}$$

Now, suppose that we can insert up to r items into the same bucket, and the probability of any of them not having set a specific bit to one is given by:

$$\left(1-\frac{1}{f}\right)^{r} \tag{2}$$

And consequently, the probability that the bit is one is:

$$1-\left(1-\frac{1}{f}\right)^{r} \tag{3}$$

Suppose the member which we want to check its membership has z number of 1's in its fingerprint. For an element membership test, if all of the array positions in the filter same as that member, are set to one, the SCFMBF claims that the element belongs to the set. The probability of this happening when the element is not part of the set is given bellow which is the false positive probability of the CSA algorithm:

$$FPP_{CSA}=\left(1-\left(1-\frac{1}{f}\right)^{r}\right)^{z} \tag{4}$$

In contrary to the Bloom Filter that had k hash functions that constructs k number of 1's in each element, in CSA algorithm, we have no information about the number of fingerprint's set bits because it's constructed by the Cuckoo hash function, so z is not constant and it can be a number from 1 to f. CSA executes the OR operation maximally for r times for each bucket. As shown in Fig. 5, it is clear that as r increases, capacity of the modified Cuckoo Filter increases but there would be a higher false positive probability unless we choose a longer fingerprint length from the beginning. Total false error probability of the SCFMBF algorithm is the multiplication of Cuckoo Filter FPP and CSA FPP.

6. FALSE ERROR PROBABILITY OF PROPOSED METHOD

Let us first derive the probability that a given set of q items collide in the same two buckets. Assume the first item x has its first bucket i_1 and a fingerprint t_x . If the other q−1 items have the same two buckets as this item x, they must have the same fingerprint t_x , which occurs with probability $\frac{1}{2^f}$ and have their first bucket either i_1 or $i_1 \oplus h(t_x)$ which occurs with probability $2/m$. Therefore, the probability of such q items sharing the same two buckets is [16]:

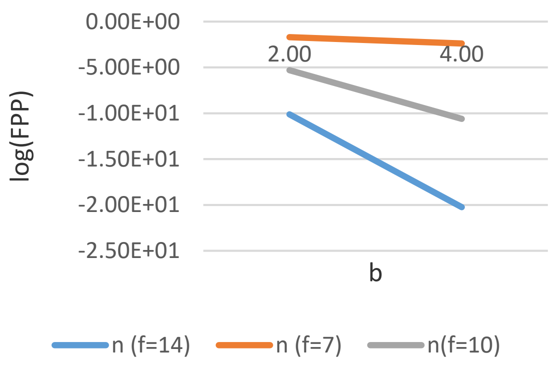

Fig. 6. False Positive Probability of Cuckoo Filter

$$\left(2/_m \times 1/_f\right)^{q-1} \tag{5}$$

The upper bound of the total probability of a false fingerprint hit is [16]:

$$1-(1-\frac{1}{2^f})^{2b} \approx 2b/_{2^f} \tag{6}$$

Now consider a construction process that inserts n random items to an empty table of $m = cn$ buckets for a constant c and constant bucket size b. whenever there are $q = 2b+1$ items mapped into the same two buckets, the insertion fails. This probability provides a lower bound for failure (and, we believe, dominates the failure probability of this construction process, although we do not prove this and do not need to in order to obtain a lower bound). Since there are in total $\binom{n}{2b+1}$ different possible sets of $2b+1$ items out of n items, the expected number of groups of $2b+1$ items colliding during the construction process is [16]:

$$FPP_{cuckoo} = \binom{n}{2b+1}\left(\frac{2}{2^f.m}\right)^{2b} \tag{7}$$

For table size=140 buckets (m=140/b) and total number of input items n=1000 and different values of n and b we have plotted the False Positive Probability values by equation 7 in Fig. 6.

Because of the great difference in FPP value when f is 14, we represented the plots in logarithmic scale. For longer fingerprints we have lower FPP. So, as we said FPP of the *SCFMBF* algorithm is calculated by the multiplication of the FPP of Cuckoo and FPP of CSA algorithm.

$$FPP_{SCFMBF} = FPP_{cuckoo} \times FPP_{CSA} \tag{8}$$

$$FPP_{SCFMBF} = \binom{n}{2b+1}\left(\frac{2}{2^f.m}\right)^{2b} \times \left(1-\left(1-\frac{1}{f}\right)^r\right)^z \tag{9}$$

We see in Fig. 6, as the fingerprint length or the bucket size increases, there is a sharp decrease in FPP. As the r (number of OR operations in CSA algorithm) increases, more insertions are possible. To cut it short, CSA algorithm increases the capacity of Cuckoo Filter. SCFMBF's capacity is the capacity of Cuckoo Filter multiplied by r. At the same time, FPP of our method is kept at a reasonable rate. The results show our method's being successful. We derived the FPP of our method by getting the average of the values and plotted them in Fig. 5. Fig. 5 shows that our method surprisingly outdistances the performance of Cuckoo Filter and maintains a lower false positive probability in every case of comparison.

Beside theoretical results, adding a counter to each bucket of the Cuckoo Filter, allowed us to detect endless loops because each time a bucket is checked, the Cuckoo Sign bit is set. When the protocol reaches a bucket with j_1=1, it realizes that a loop has happened and it cuts the search. Also, Cuckoo counter adds deletion capability to Bloom Filter by counting the number of added items and subtracting the removed items.

7. CONCLUSION

In our method we tried to eliminate Cuckoo Filter's limitations. Cuckoo Filter's endless loop problem is solved by the endless loop algorithm, surprisingly there was no need for extra capacity because we benefit from using small counters and a single bit for every bucket and by our CSA algorithm Cuckoo Filter is able to handle more insertions, the idea of CSA was inspired from the Bloom Filter's basic logical operation (OR). The false positive probability according to the derived mathematical equation, has been improved a lot. We have studied the relation between SCFMBF's parameters (same as Cuckoo Filter) and false positive probability. In Fig. 5, CSA algorithm's false positive probability is plotted with different f (fingerprint length) and r (number of 1's in the fingerprint) values. As r decreases, FPP plot falls. As the fingerprint length increases, FPP decreases. For table size=140 buckets (m=140/b) and total number of input items n=1000 and different values of n and b the False Positive Probability of Cuckoo Filter is plotted in Fig. 6. For longer fingerprints FPP has lower values, but long fingerprint length needs more storage space. Fig. 7 shows the importance of parameter choice. Larger bucket size results in less FPP. We

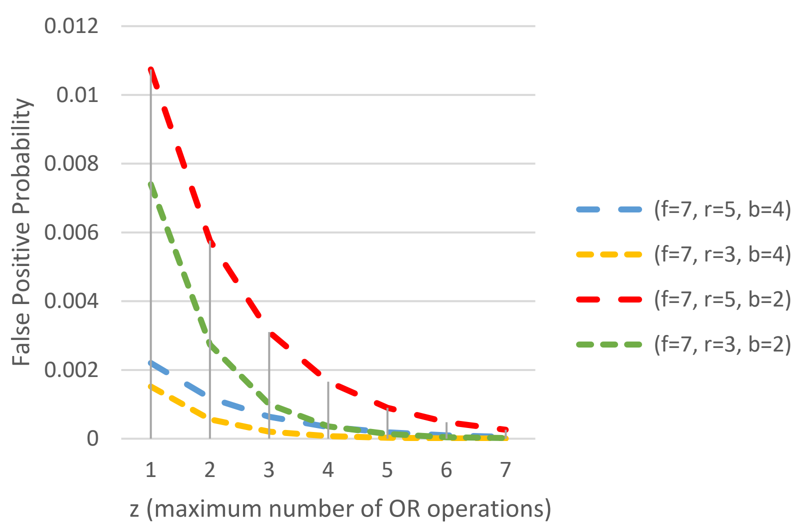

Fig. 7. False Positive Probability of SCFMBF

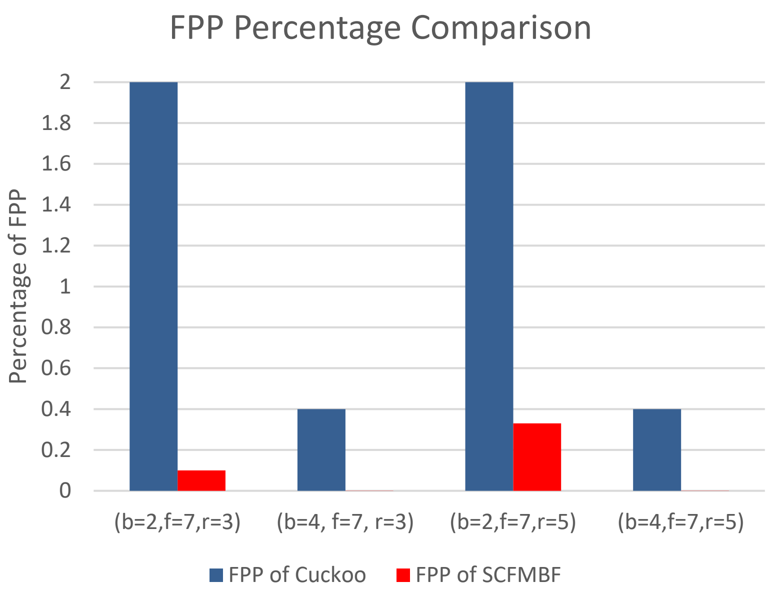

Fig. 8. FPP Percentage Comparison

finally compared SCFMBF with Cuckoo Filter for f=7, b=2,4 and r=3,5 (Fig. 8). In every case of study, there is a lot of improvement. FPP of SCFMBF in the worst case is four times less than the FPP of Cuckoo Filter that means we have also improved the capacity Cuckoo Filter to a great extent. The aim of the protocol is fulfilled.

In the future works this algorithm can be used in some applications such as cloud storage integrity check, network and different set membership problems.

REFERENCES

[1] P. Brass, Advanced Data Structures, Cambridge University Press, (2008) 402–405.

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

[3] J.K. Mullin, D.J. Margoliash, A tale of three spelling checkers, Software, Practice and Experience, (1990) 625–630.

[4] S. Czerwinski, B.Y. Zhao, T. Hodes, A.D. Joseph, R. Katz, An Architecture for a Secure Service Discovery Service, Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, ACM Press, (1999) 24–35.

[5] J. Wang, A survey of web caching schemes for the internet, ACM SIGCOMM Computer Communication Review, (1999).

[6] R.P. Laufer, P.B. Velloso, D.d.O. Cunha, I.M. Moraes, M.D.D. Bicudo, M.D.D. Moreira, O.C.M.B. Duarte, Towards stateless singlepacket IP traceback, Proceedings of the 32nd IEEE Conference on Local Computer Networks, (2007) 548–555.

[7] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Transactions on Networking, 28(4) (1998) 254–265.

[8] M. Xiao, Y. dai, X. Luo, A Split Bloom Filter for Better Performance, Journal of Applied Security Research, 15(2) (2019)1–14.

[9] D. Guo, J. Wu, H. Chen, Y. Yuan, X. Luo, The dynamic Bloom filters, IEEE Transactions on knowledge and data engineering, 22(1) (2010)

120–133.
[10] S. Geravand, M. Ahmadi, A novel adjustable matrix Bloom filterbased copy detection system for digital libraries, Proceedings of IEEE International Conference on Computer and Information Technology, (2011).
[11] M. Mitzenmacher, Compressed Bloom filters, Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, (2001) 144–150.
[12] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, G. Varghese, Bloom filters via d-left hashing and dynamic bit reassignment, 44th Allerton Conference, (2006).
[13] K. Shanmugasundaram, H. Bronnimann, N. Memon, Payload attribution via hierarchical Bloom Filter, Proceedings of the 11th ACM conference on Computer and communicationsecurity (CCS 04), (2004) 31–41.
[14] S. Cohen, Y. Matias, Spectral Bloom filters, SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. New York, NY, USA, ACM, (2003) 241–252.
[15] Y. Matsumoto, H. Hazeyama, Y. Kadobayashi, Adaptive Bloom filter: A space-efficient counting algorithm for unpredictable network traffic, IEICE Trans. Inf. Syst., E91-D(5) (2008)1292–1299.
[16] P.S. Almeida, C. Baquero, N. Preguic, D. Hutchison, Scalable Bloom filters, Inf. Process. Lett., 101(6) (2007) 255–261.
[17] J. Bruck, J. Gao, A. Jiang, Weighted Bloom filter, IEEE International Symposium on Information Theory (ISIT'06), (2006).
[18] M. A. Bender, M. F. Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillanne, E. Zadoc, Don't Thrash: How to Chache you Hash on Flash, Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HOTStorage11), (2012).
[19] B. Fan, D. Andersen, M. Kaminsky, M. Mitzenmacher, Cuckoo filter: practically better than Bloom, Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT 14), (2014) 75–88.
[20] J. Grashofer, F. Jacob, H. Hartenstein, Towards Application of Cuckoo Filters in Network Security Monitoring, 14th international conference on network and service management (CNSM), (2018).
[21] M. Mitzenmacher, S. Pontarelli, P. Reviriego, Adaptive Cuckoo filters, arXiv preprint, (2017).
[22] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, P. Zue, SmartCuckoo: A Fastand Cost-Efficient Hashing Index Scheme for Cloud Storage Systems, USENIX Annual Technical Conference, (2017) 553–565.
[23] Kirsch, A. Mitzenmacher, U. Wieder, More Robust Hashing: Cuckoo Hashing with a Stash, SIAM Journal on Computing, 39(4), (2009) 1543–1561.
[24] U. Erlingsson, F. Mcsherry, M. Manasse, A cool and practical alternative to traditional hash tables, Proceedings of the Seventh Workshop on Distributed Data and Structures (WDAS), (2006).
[25] B. Fan, D.G. Andersen, M. Kaminsky, MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing, Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation, (2013).
[26] A. Singh, S. Garg, K. Kaur, K. KwangRaymond Cool, Fuzzy-folded Bloom Filter-as-a-Service for Big Data Storage in the Cloud, IEEE Transactions on Industrial Informatics, (2018) 1–10.
[27] K. Sasaki, S. Sugiura, S. Makido, N. Suzuki, Bloom-Filter Aided Two-Layered Structured Overlay for Highly-Dynamic Wireless Distributed Storage, IEEE Communications Letters, 17(4) (2013) 629–632.
[28] C.E. Rothenberg, C.A.B. Macapuna, F.L. Verdi, M.F. Magalhaes, The Deletable Bloom Filter: A New Member of the Bloom Family, IEEE Communications Letters, 14(6) (2010) 557–559.
[29] P. Reviriego, S. Pontarelli, J. A. Maestro, M. Ottavi, A Synergetic Use of Bloom Filters for Error Detection and Correction, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 23(3) (2015) 584–587.
[30] H. Lim, J. Lee, H. Byun, C. Yim, Ternary Bloom Filter Replacing Counting Bloom Filter, Communications Letters, 21(2) (2016), 278-281.
[31] D. Ficara, A.D. Pietro, S. Giordano,G. Procissi, F. Vitucci, Enhancing
Counting Bloom Filters Through Huffman-Coded Multilayer, IEEE/ ACM Transactions on Networking, 18(6) (2010), 1977-1987.
[32] Kumar, J. Xu, J. Wang, Space-Code Bloom Filter for Efficient
Per-Flow Traffic Measurement, IEEE Journal on Selected Areas in Communications, 24(12) (2006) 2327-2339.
[33] F. Hao, M. Kodialam, T. V. Lakshman, H. Song, Fast Dynamic Multiple-Set Membership Testing Using Combinatorial Bloom Filters, IEEE/ACM Transactions on Networking, 20(1) (2012) 295–304.
[34] H. Duan, S. Yu, M. Mei, W. Zhan, L. Li, Cstore: a desktop-oriented
distributed public cloud storage system, Computers & Electrical Engineering, 42 (2015) 60–73.
[35] S. Geravand, M. Ahmadi, A novel adjustable matrix bloom filter-based copy detection system for digital libraries, IEEE 11th International Conference on Computer and Information Technology(CIT), (2011) 518–525.
[36] S. Xiong, F. Wang, Q. Cao, A bloom filter based scalable data integrity check tool for large-scale dataset, First Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems(PDSW-DISCS), (2016) 55–60.
[37] Y. Qiao, T. Li, S. Chen, Fast bloom filters and their generalization, IEEE Transactions on Parallel and Distributed Systems, 25 (2013), 93–103.
[38] P. Reviriego, K. Christensen, J. A. Maestro, A comment on "fast Bloom Filters and their generalization", IEEE Transactions on Parallel and Distributed Systems, 27 (2015) 303–304.
[39] A. Crainiceanu, D. Lemire, Bloofi: Multidimensional Bloom Filters, Information Systems, 54 (2015), 311–324.
[40] A. Singh, S. Garg, S. Batra, N. Kuma, J.J. Rodrigues, Bloom Filter based optimization scheme for massive data handling in IoT environment, Future Generation Computer Systems, 82 (2018) 440–449.
[41] F. Grandi, On the analysis of Bloom Filters, Information Processing Letters, 129 (2018) 35–39.

HOW TO CITE THIS ARTICLE

Ha. Sasaniyan Asl, B. Mozaffari Tazehkand, M.J. Museviniya, A Cuckoo Filter Modification Inspired by Bloom Filter, AUT J. Elec. Eng., 51(2) (2019) 187-200.

DOI: 10.22060/eej.2019.16370.5287